JS

// Learn JavaScript
// step by step
// to build amazing things
</>

JavaScript step S

FROM BEGINNER
TO ADVANCED

- </> CLEAR EXPLANATIONS
- { } PRACTICAL EXAMPLES
- >_ REAL PROJECTS
- ★ MODERN JAVASCRIPT ES6+ AND BEYOND

</>
STEP BY STEP
LEARN CODE
BUILD THE
FUTURE

FEDI YAHIYAOUI

JavaScript Step S

From Beginner to Master

A Complete Modern Guide to JavaScript Programming

Fedi Yahiyaoui

First Edition · 2026

JavaScript Step S: From Beginner to Master

A Complete Modern Guide to JavaScript Programming

First Edition, 2026

Copyright © 2026 Fedi Yahiyaoui. All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law.

All code examples in this book are released for personal and educational use. You may adapt them freely in your own projects.

JavaScript is a trademark of Oracle Corporation. This book is an independent educational work and is not affiliated with, endorsed by, or sponsored by Oracle or any browser vendor.

Typeset in XCharter and Inconsolata with LaTeX.
Cover design and interior design by Fedi Yahiyaoui.

To every curious mind who ever opened a browser console,
typed their first `console.log("hello")`*,*
and felt the small, electric joy of making a machine listen.

This book is for you.

— Fedi Yahiyaoui

Preface

Every professional developer you admire once sat exactly where you are sitting now: in front of a blank editor, wondering where to begin. I know, because that was me. My name is Fedi, and this book is the guide I wish someone had placed in my hands when I wrote my first line of JavaScript.

JavaScript is a remarkable language. It is the only programming language that runs natively in every web browser on the planet, which means it is also the most accessible language ever created: you already own everything you need to run it. Yet JavaScript is far more than a scripting toy. It powers servers, mobile apps, desktop software, games, robots, and the interfaces of almost every product you use daily. Learning it deeply is one of the highest-leverage investments you can make in yourself.

Why “Step S”?

The letter S has followed this project from the beginning — the curve of the S is the shape of the learning journey itself. It starts slow, bends through difficulty, and straightens into mastery. The book is organized into four tiers — **Beginner**, **Intermediate**, **Advanced**, and **Master** — so that every chapter stands on ground you have already walked.

What makes this book different

Most tutorials teach syntax. This book teaches *thinking*. Every module follows a consistent rhythm: concepts explained from first principles, real code you can run immediately, the mistakes beginners actually make (and how to avoid them), exercises at four difficulty levels, quizzes, and interview questions drawn from real hiring processes. Then, in the capstone section, you build twelve complete projects — from a greeting bot to a visual pathfinding engine — that weave everything together.

A word on honesty: no book can make you a master by reading alone. Mastery lives in your fingers, not your eyes. Type every example. Break every example. Fix it again. If you do that, these roughly three hundred pages will take you further than you currently believe possible.

Acknowledgments

To the open-source community that documents, answers, and teaches without asking anything in return — this book stands on your shoulders. And to you, the reader: thank you for trusting me with your learning journey. I do not take it lightly.

Fedi Yahiyaoui
2026

How to Use This Book

Who this book is for

This book assumes **zero programming experience**. If you have never written a line of code, start at Module 1 and proceed in order. If you already know some JavaScript, use the self-check quiz in Module 7 to find your level, then jump to the tier that challenges you.

The structure of every module

Each of the 25 modules follows the same professional rhythm:

1. **Learning objectives** — what you will be able to do afterwards.
2. **Core concepts** — explained from first principles, with diagrams.
3. **Code examples** — modern, runnable JavaScript, each one explained.
4. **Common mistakes** — the errors real beginners make, before you make them.
5. **Best practices and pro tips** — how professionals write the same code.
6. **Mini challenge** — a small problem to solve immediately.
7. **Exercises** — four levels, from warm-up to professional.
8. **Quiz** — fast self-assessment.
9. **Summary** — the module on one page.
10. **Interview questions** — what employers ask about this topic.

The boxes

Throughout the book you will meet colored boxes. Each color has a fixed meaning:

PRO TIP
Shortcuts, tricks, and professional habits that save you time.

COMMON MISTAKE
A mistake so common it deserves its own alarm. Read these twice.

NOTE
Background context, historical notes, and clarifications.

MINI CHALLENGE
Stop reading. Solve this before continuing.

BEST PRACTICE
The way professionals write this code in real teams.

INTERVIEW CORNER

Questions from real technical interviews, with guidance.

Prerequisites

You need three things, all free:

- **A computer** running Windows, macOS, or Linux.
- **A modern browser** — Chrome, Edge, or Firefox. This is your JavaScript engine.
- **A code editor** — Visual Studio Code is recommended and used throughout.

  No prior programming, no accounts, no paid tools.

Setting up your environment (5 minutes)

1. Install **Visual Studio Code** from `code.visualstudio.com`.
2. Install your browser of choice (you almost certainly already have one).
3. Open your browser's console right now: press **F12** (or **Ctrl+Shift+J** on Windows/Linux, **Cmd+Option+J** on macOS).
4. Type the following and press Enter:

```
console.log("Hello, Fedi!");
```

If you saw the greeting appear, congratulations — your development environment is ready, and you have already run your first JavaScript program.

PRO TIP

For most of Tier 1 and Tier 2, the browser console is all you need. From Module 9 (DOM) onward, you will work with `.html` files opened directly in the browser. No servers, no build tools, no configuration — by design.

A personal note on pacing

Do not rush. One module per day is an excellent pace; one per week is perfectly fine. The exercises are not optional decoration — they are where the learning actually happens. When you finish the final project (the Fedi Pathfinder), you will have a portfolio of twelve working applications and the confidence to call yourself a JavaScript developer.

Let us begin.

Contents

Preface v

How to Use This Book vii

I Tier 1 — Beginner 1

1 Variables and Strings 3
- 1.1 What Is a Variable? . . . 3
- 1.2 `var`, `let`, and `const` . . . 4
  - 1.2.1 `const` — the default choice . . . 4
  - 1.2.2 `let` — when the value must change . . . 4
  - 1.2.3 `var` — the historical keyword . . . 4
- 1.3 Naming Conventions . . . 5
- 1.4 String Basics . . . 5
- 1.5 String Concatenation . . . 6
- 1.6 Template Literals . . . 7
- 1.7 Common String Methods . . . 8
  - 1.7.1 Changing case . . . 8
  - 1.7.2 Searching inside strings . . . 8
  - 1.7.3 Extracting pieces: `slice` . . . 8
  - 1.7.4 Cleaning and replacing . . . 9
  - 1.7.5 Splitting strings into arrays . . . 9
- 1.8 String Immutability . . . 9
- 1.9 Exercises . . . 10
- 1.10 Quiz . . . 10

2 Booleans and Numbers 12
- 2.1 Boolean Values . . . 12
- 2.2 Comparison Operators . . . 12
  - 2.2.1 Always use strict equality . . . 13
- 2.3 Logical Operators . . . 13
  - 2.3.1 Short-circuit evaluation . . . 14
- 2.4 Truthy and Falsy . . . 14
- 2.5 Number Basics . . . 15
  - 2.5.1 Compound assignment and increment . . . 16
  - 2.5.2 Special numeric values . . . 16
- 2.6 The Math Object . . . 16

2.6.1 Random numbers 16
2.7 Precision and Rounding 17
2.8 Type Conversion 18
2.9 Exercises 18
2.10 Quiz 19

3 Functions **20**
3.1 Declaring Functions 20
3.2 Return Values 21
3.3 Function Expressions 21
3.4 Arrow Functions 22
3.5 Default Parameters 23
3.6 Scope 23
3.6.1 Shadowing 24
3.6.2 Global vs local: why globals are dangerous 24
3.7 Introduction to Closures 24
3.8 Exercises 25
3.9 Quiz 25

4 Arrays and Memory Optimization **27**
4.1 Creating Arrays and Static Builders 27
4.2 Indexing and Length 28
4.3 V8 Engine Optimization: Packed vs Holey Arrays 28
4.4 Adding and Removing Elements 28
4.5 Slicing and Splicing 29
4.6 Modern Non-Mutating Array APIs 29
4.7 Multidimensional Arrays 29
4.8 Array Destructuring and Spread Operator 30
4.9 Exercises 30
4.10 Quiz 31

5 Objects and Prototypes **32**
5.1 Object Literals and Computed Properties 32
5.2 Accessing Properties: Dot, Bracket, and Optional Chaining 33
5.3 Methods, `this`, and Explicit Binding 33
5.4 Object References and Heap Memory 34
5.5 Property Descriptors and Getters/Setters 35
5.6 Object Mutation Controls 36
5.7 The Prototype Chain 36
5.8 Exercises 36
5.9 Quiz 37

6 Loops and Iteration **38**
6.1 The Classic `for` Loop . . . 38
6.2 Condition-Driven Iteration: `while` and `do...while` . . . 39
6.3 Modern Iteration: `for...of` vs `for...in` . . . 39
6.4 Labeled Statements and Controlling Loop Flow . . . 40
6.5 Generators and Lazy Evaluation . . . 41
6.6 Nested Loops and Performance Pitfalls . . . 41
6.7 Exercises . . . 42
6.8 Quiz . . . 42

7 JavaScript Fundamentals Review **44**
7.1 Core Concepts Architecture . . . 44
7.2 The Top 10 Beginner Traps and Anti-Patterns . . . 44
7.2.1 Trap 1: Accidental String Concatenation with Numbers . . . 44
7.2.2 Trap 2: Mutating Objects by Shared Reference . . . 45
7.2.3 Trap 3: Testing Array Emptiness with Truthiness . . . 45
7.2.4 Trap 4: Loose Equality Coercion . . . 45
7.2.5 Trap 5: Forgetting to Return from a Function . . . 45
7.2.6 Trap 6: Block Scope vs Function Scope (var hoisting) . . . 46
7.2.7 Trap 7: Lexical "this" in Arrow Methods . . . 46
7.2.8 Trap 8: Modifying Array Elements During Iteration . . . 46
7.2.9 Trap 9: Variable Shadowing Confusion . . . 47
7.2.10 Trap 10: Floating Point Precision Errors . . . 47
7.3 Synthesis Project: The Fedi Mini E-Commerce Engine . . . 47
7.4 Tier 1 Diagnostic Self-Check Exam . . . 48

II Tier 2 — Intermediate **50**

8 Higher Order Functions and Callbacks **52**
8.1 First-Class Functions and Callbacks . . . 52
8.2 Array Transformations: Map, Filter, and Reduce . . . 53
8.3 Aggregating with `reduce()`: The Initial Value Trap . . . 53
8.4 Searching: `find`, `findIndex`, `some`, and `every` . . . 54
8.5 Sorting Arrays: Mutating vs Non-Mutating APIs . . . 54
8.6 Chaining Array Pipelines . . . 55
8.7 Exercises . . . 55
8.8 Quiz . . . 56

9 DOM Manipulation and Events **57**
9.1 What Is the DOM? . . . 57
9.2 Selecting Elements: Live Collections vs Static Lists . . . 57
9.3 Changing Content, Attributes, and Classes . . . 58
9.4 Creating and Appending with Document Fragments . . . 59
9.5 Event Propagation: Capture, Target, and Bubble . . . 59

9.6 Event Delegation and Cancellation 60
9.7 Event Optimization: Debounce and Throttle 60
9.8 Custom Events 61
9.9 Modern Observers: Intersection and Mutation 62
9.10 Exercises 63
9.11 Quiz 63

10 Web Accessibility **64**
10.1 Why Accessibility Matters 64
10.2 Semantic HTML: The Foundation of Accessibility 64
10.3 ARIA Attributes: Enhancing Custom Widgets 65
10.4 Keyboard Navigation and Focus Management 66
10.5 Live Regions for Dynamic Content 67
10.6 Color Contrast and Visual Accessibility 68
10.7 Accessible Forms 68
10.8 Testing Accessibility 69
10.9 Exercises 69
10.10Quiz 70

11 Debugging and Error Handling **71**
11.1 Reading Error Messages and Stack Traces 71
11.2 Exception Propagation and the Call Stack 72
11.3 The try...catch...finally Statement 72
11.4 Asynchronous Error Handling 73
11.5 Creating Custom Error Hierarchies 74
11.6 Advanced Console Diagnostics 75
11.7 Browser DevTools, Breakpoints, and Source Maps 75
11.8 Profiling Memory Leaks and Detached DOM Nodes 76
11.9 Exercises 77
11.10Quiz 77

12 Regular Expressions **78**
12.1 Creating Regular Expressions 78
12.2 Character Classes and Shorthands 79
12.3 Quantifiers and Boundaries 79
12.4 Regex Flags in Depth 79
12.5 String Methods Supporting Regex 80
12.6 Named Capture Groups and Backreferences 80
12.7 Lookahead and Lookbehind Assertions 81
12.8 Catastrophic Backtracking and Regex Performance 81
12.9 Exercises 82
12.10Quiz 82

13 Form Validation **84**
13.1 HTML5 Built-In Validation Attributes . . . 84
13.2 The Constraint Validation API . . . 85
13.3 Real-Time Inline Validation . . . 86
13.4 Cross-Field Validation: Password Confirmation . . . 87
13.5 Form Submission with Validation Gate . . . 87
13.6 Input Sanitization and XSS Prevention . . . 88
13.7 Exercises . . . 88
13.8 Quiz . . . 89

14 Dates and Time **90**
14.1 Creating Date Objects . . . 90
14.2 Reading Date Components . . . 91
14.3 UTC vs Local Time . . . 91
14.4 Date Arithmetic and Durations . . . 92
14.5 The Intl.DateTimeFormat API . . . 92
14.6 Relative Time with Intl.RelativeTimeFormat . . . 93
14.7 Practical Pattern: Countdown Timer . . . 94
14.8 Exercises . . . 94
14.9 Quiz . . . 95

15 Audio, Video, and the Web Audio API **96**
15.1 The HTMLMediaElement API . . . 96
15.2 Media Events . . . 97
15.3 Building a Custom Audio Player . . . 97
15.4 The Web Audio API: Synthesis and Analysis . . . 98
15.5 Audio Frequency Visualization with Canvas . . . 99
15.6 Picture-in-Picture API . . . 100
15.7 Recording Audio with MediaRecorder . . . 100
15.8 Exercises . . . 101
15.9 Quiz . . . 101

16 Maps and Sets **103**
16.1 The Map Data Structure . . . 103
16.2 Map vs Plain Object: When to Use Which . . . 104
16.3 Object Keys with Maps . . . 104
16.4 Iterating Maps . . . 105
16.5 The Set Data Structure . . . 105
16.6 Set-Theory Operations . . . 106
16.7 WeakMap and WeakSet: Memory-Safe References . . . 106
16.8 Exercises . . . 107
16.9 Quiz . . . 107

17 localStorage and CRUD Operations **109**
17.1 The Browser Storage Landscape . . . 109
17.2 The localStorage API . . . 109
17.3 Serialization Boundaries and Custom JSON Replacers . . . 110
17.4 Quota Limits and QuotaExceededError . . . 111
17.5 Implementing the CRUD Cycle . . . 112
17.6 Cross-Tab State Synchronization . . . 113
17.7 Exercises . . . 114
17.8 Quiz . . . 114

III Tier 3 — Advanced **115**

18 Classes and Object-Oriented Programming **117**
18.1 Introduction to Classes . . . 117
18.2 Private Fields, Encapsulation, and the "in" Operator . . . 118
18.3 Getters, Setters, and Private Statics . . . 119
18.4 Inheritance and Abstract Classes . . . 119
18.5 Class Mixins and Multiple Inheritance . . . 120
18.6 Prototypal Inheritance and Memory Optimization . . . 121
18.7 Exercises . . . 122
18.8 Quiz . . . 122

19 Recursion **124**
19.1 The Anatomy of Recursion . . . 124
19.2 Tracing the Call Stack . . . 125
19.3 Classic Recursive Algorithms . . . 125
19.4 Processing Hierarchical Data . . . 126
19.5 Stack Overflow and Maximum Call Stack Size . . . 127
19.6 Tail Call Optimization (TCO) . . . 127
19.7 Recursion vs Iteration . . . 128
19.8 Exercises . . . 128
19.9 Quiz . . . 129

20 Data Structures **130**
20.1 The Stack (LIFO) . . . 130
20.2 The Queue (FIFO) . . . 131
20.3 Linked Lists: Singly and Doubly . . . 132
20.4 Binary Search Trees (BST) . . . 133
20.5 Hash Tables and Collision Resolution . . . 134
20.6 Exercises . . . 135
20.7 Quiz . . . 136

21 Algorithms and Big O Notation **137**
21.1 Understanding Big O Notation . . . 137
21.2 Searching Algorithms . . . 138
21.3 Sorting Algorithms . . . 138
21.4 The Two-Pointer Pattern: Opposite and Fast/Slow . . . 139
21.5 The Sliding Window Pattern: Fixed and Dynamic . . . 140
21.6 Exercises . . . 141
21.7 Quiz . . . 141

IV Tier 4 — Master **143**

22 Graphs and Trees **145**
22.1 Tree Fundamentals and Terminology . . . 145
22.2 Binary Tree Node Implementation . . . 146
22.3 Tree Traversal Algorithms . . . 146
22.4 Binary Search Tree Operations . . . 147
22.5 Graph Fundamentals and Terminology . . . 148
22.6 Graph Representation: Adjacency List . . . 148
22.7 Breadth-First Search (BFS) . . . 149
22.8 Depth-First Search (DFS) . . . 150
22.9 Exercises . . . 151
22.10 Quiz . . . 151

23 Dynamic Programming **153**
23.1 When to Apply Dynamic Programming . . . 153
23.2 Strategy 1: Top-Down Memoization . . . 154
23.3 Strategy 2: Bottom-Up Tabulation . . . 154
23.4 Space Optimization: Rolling Variables . . . 155
23.5 Classic DP Problem: Climbing Stairs . . . 155
23.6 Classic DP Problem: Coin Change . . . 156
23.7 Classic DP Problem: Unique Grid Paths . . . 156
23.8 Classic DP Problem: Longest Common Subsequence . . . 157
23.9 Exercises . . . 158
23.10 Quiz . . . 158

24 Functional Programming **159**
24.1 Pure Functions . . . 159
24.2 Immutability: Never Mutate, Always Copy . . . 160
24.3 Function Composition . . . 161
24.4 Currying . . . 161
24.5 Declarative vs Imperative Style . . . 162
24.6 Closures as Functional Data Encapsulation . . . 163
24.7 Exercises . . . 163
24.8 Quiz . . . 164

25 Asynchronous JavaScript **165**
25.1 The Event Loop Architecture . . . 165
25.2 From Callbacks to Promises . . . 166
25.3 Modern `async` and `await` . . . 167
25.4 The Fetch API: HTTP Communication . . . 167
25.5 Cancelling Requests with AbortController . . . 168
25.6 Concurrent Promise Combinators . . . 168
25.7 Exercises . . . 169
25.8 Quiz . . . 169

V Build with Fedi — Capstone Projects **171**

26 Project 1: Fedi Greeting Bot **173**
26.1 Project Overview and Goals . . . 173
26.2 User Interface Architecture and HTML . . . 173
26.3 Styling the Component with CSS . . . 174
26.4 JavaScript Implementation and Logic . . . 174
26.5 Step-by-Step Code Analysis . . . 176
26.6 Testing Checklist and Extensions . . . 176

27 Project 2: Fedi Digital Pet Game **177**
27.1 Project Requirements and State Machine . . . 177
27.2 HTML Layout Structure . . . 177
27.3 CSS Component Styling . . . 178
27.4 JavaScript State and Game Engine . . . 179
27.5 Testing Checklist and Extensions . . . 180

28 Project 3: Fedi Flashcard Quiz App **181**
28.1 Project Overview and Requirements . . . 181
28.2 HTML Layout Structure . . . 181
28.3 CSS 3D Flip Stylesheet . . . 182
28.4 JavaScript Flashcard Engine . . . 183
28.5 Testing Checklist and Extensions . . . 185

29 Project 4: Fedi Motorcycle Shop **186**
29.1 Project Requirements and Architecture . . . 186
29.2 HTML Catalog Layout . . . 186
29.3 CSS Grid Stylesheet . . . 187
29.4 JavaScript Catalog and Cart Engine . . . 187
29.5 Testing Checklist and Extensions . . . 189

30 Project 5: Fedi Drum Machine **190**
30.1 Project Overview and Requirements . . . 190
30.2 HTML Pad Matrix Layout . . . 190

30.3 CSS Pad Visualizer Stylesheet . . . 191
30.4 JavaScript Audio Controller . . . 191
30.5 Testing Checklist and Extensions . . . 192

31 Project 6: Fedi Travel Weather Planner 193
31.1 Project Overview and Requirements . . . 193
31.2 HTML Dashboard Structure . . . 193
31.3 CSS Dashboard Layout . . . 194
31.4 JavaScript Async Weather Engine . . . 194
31.5 Testing Checklist and Extensions . . . 196

32 Project 7: Fedi Expense Tracker 197
32.1 Project Overview and Requirements . . . 197
32.2 HTML Layout Structure . . . 197
32.3 CSS Component Stylesheet . . . 198
32.4 Testing Checklist and Extensions . . . 200

33 Project 8: Fedi Password Generator 201
33.1 Project Overview and Requirements . . . 201
33.2 HTML Generator Layout . . . 201
33.3 CSS Security Console Stylesheet . . . 202
33.4 JavaScript Cryptographic Engine . . . 203
33.5 Testing Checklist and Extensions . . . 204

34 Project 9: Fedi Movie Search 205
34.1 Project Overview and Requirements . . . 205
34.2 HTML Layout Structure . . . 205
34.3 CSS Cinema Grid Stylesheet . . . 205
34.4 JavaScript Debounce and Search Engine . . . 206
34.5 Testing Checklist and Extensions . . . 207

35 Project 10: Fedi Task Manager with Classes 209
35.1 Project Architecture and Class Design . . . 209
35.2 HTML Kanban Board Structure . . . 209
35.3 CSS Kanban Board Stylesheet . . . 210
35.4 Testing Checklist and Extensions . . . 213

36 Project 11: Nth Fibonacci Number Visualizer 214
36.1 Project Architecture and Complexity Laboratory . . . 214
36.2 HTML Layout Structure . . . 214
36.3 CSS Laboratory Stylesheet . . . 215
36.4 Testing Checklist and Extensions . . . 217

37 Project 12: Fedi Pathfinder **218**
37.1 Project Overview and Requirements . . . 218
37.2 HTML Grid and Controls Layout . . . 218
37.3 CSS Pathfinding Grid Stylesheet . . . 219
37.4 Testing, Diagnostics, and Capstone Mastery . . . 222

VI JavaScript Step S — Master Review **223**

38 Master Review and Progression Map **224**
38.1 The Four Tiers of JavaScript Mastery . . . 224
38.1.1 Tier 1: Beginner — The Language Fundamentals . . . 224
38.1.2 Tier 2: Intermediate — The Modern Web Platform . . . 224
38.1.3 Tier 3: Advanced — Software Engineering & Algorithms . . . 224
38.1.4 Tier 4: Master — Concurrency, Architecture & Scale . . . 225

39 Syntax and Methods Cheat Sheets **226**
39.1 Variables and Operators . . . 226
39.2 Essential Array Methods . . . 226
39.3 Essential String Methods . . . 227
39.4 DOM Manipulation and Events . . . 227
39.5 Asynchronous JavaScript . . . 227

40 Technical Interview Questions **228**
40.1 Core Language and Runtime . . . 228
40.2 DOM and Web APIs . . . 228
40.3 Algorithms and Data Structures . . . 229

41 Advanced Project Ideas for Master Practice **230**
41.1 Tier 1: Beginner Portfolio Additions . . . 230
41.1.1 Project 1: The Personal Finance CLI Ledger . . . 230
41.1.2 Project 2: Fedi's Task Automator Library . . . 231
41.2 Tier 2: Intermediate Dynamic Web Applications . . . 231
41.2.1 Project 3: The Interactive Recipe Book with LocalStorage CRUD . . . 231
41.2.2 Project 4: The Accessible Markdown Editor . . . 231
41.3 Tier 3: Advanced Architecture and Algorithms . . . 231
41.3.1 Project 5: Event-Driven E-Commerce Cart Engine . . . 231
41.3.2 Project 6: Visual Call Stack Simulator . . . 232
41.4 Tier 4: Master Concurrency and Optimization . . . 232
41.4.1 Project 7: Asynchronous GitHub Dashboard with Retry/Timeout . . . 232
41.4.2 Project 8: Dynamic Programming Grid Path Visualizer . . . 232

42 Glossary of Terms **234**

43 Conclusion: The Journey Ahead **236**
43.1 The Senior Engineer's Mindset 236
43.2 From Beginner to Master: The Self-Assessment Checklist 236
43.3 Where to Go Next 237

Index **239**

List of Figures

3.1 Nested scopes: inner scopes can see outward, never inward. 23

4.1 V8 Array Elements Optimization Transitions 28

5.1 Variables Storing Pointers to the Same Object in Heap Memory 35
5.2 The Prototype Inheritance Chain 36

6.1 Lifecycle and Control Flow of a Classic `for` Loop 39

7.1 The Interlocking Architecture of JavaScript Fundamentals 44

8.1 Visualizing Map, Filter, and Reduce Operations 53

9.1 The Hierarchical Structure of the DOM Tree 58
9.2 DOM Event Propagation Flow 59

11.1 Exception Bubbling Up the Call Stack 72
11.2 The DevTools Execution Stepping Interface 75

12.1 Finite State Automaton Diagram for a Simple Pattern 82

15.1 Web Audio API Signal Processing Graph 98

17.1 Browser Client-Side Storage Architecture 110

18.1 The Prototype Chain Lookup Mechanism 121

19.1 Call Stack Frames During Recursive Countdown 125

20.1 Singly Linked List Chain Architecture 132
20.2 A Balanced Binary Search Tree (BST) 133
20.3 Hash Table with Separate Chaining Collision Resolution 134

21.1 Comparison of Common Big O Complexity Curves 137
21.2 Quick Sort Partitioning Concept 139

22.1 A Balanced Binary Search Tree with Labeled Terminology 145
22.2 An Undirected Graph with 5 Vertices and 6 Edges 148

23.1 Naive Fibonacci Recursion Tree Showing Overlapping Subproblems 153

25.1 The JavaScript Event Loop and Concurrency Lifecycle 166

41.1 Scan the QR Code to Visit the GitHub Solutions Repository 230

List of Code Listings

1.1 Your first variable . . . 3
1.2 Changing a variable's value . . . 3
1.3 Constants cannot be reassigned . . . 4
1.4 A value that legitimately changes . . . 4
1.5 Why var surprises beginners . . . 4
1.6 Names that explain themselves . . . 5
1.7 Three ways to write a string . . . 5
1.8 Quotes inside strings . . . 6
1.9 The length property . . . 6
1.10 Indexing into a string . . . 6
1.11 Joining strings with + . . . 6
1.12 Concatenation fatigue . . . 7
1.13 Template literals: readable string building . . . 7
1.14 Expressions inside placeholders . . . 7
1.15 Multi-line strings . . . 7
1.16 toUpperCase and toLowerCase . . . 8
1.17 includes, indexOf, startsWith, endsWith . . . 8
1.18 Slicing strings . . . 8
1.19 Combining indexOf and slice . . . 9
1.20 trim, replace, replaceAll, repeat . . . 9
1.21 split: from string to array . . . 9
1.22 Strings never change in place . . . 9
2.1 The two boolean values . . . 12
2.2 The comparison operators . . . 12
2.3 === vs == . . . 13
2.4 AND, OR, NOT . . . 13
2.5 Combining conditions like a professional . . . 14
2.6 Short-circuiting in practice . . . 14
2.7 Testing truthiness . . . 15
2.8 Arithmetic operators . . . 15
2.9 Modulo in the wild . . . 15
2.10 Updating numbers concisely . . . 16
2.11 Infinity and NaN . . . 16
2.12 Everyday Math methods . . . 16
2.13 The standard random-integer recipe . . . 16
2.14 Floating-point reality . . . 17
2.15 Working around float imprecision . . . 17

2.16 String to number: the reliable ways . . . 18
2.17 Number to string . . . 18
2.18 Coercion traps . . . 18
3.1 A function declaration . . . 20
3.2 With and without return . . . 21
3.3 Guard clauses: fail fast, stay flat . . . 21
3.4 A function as a value . . . 21
3.5 Hoisting in action . . . 22
3.6 Arrow function syntax, step by step . . . 22
3.7 Default parameters . . . 23
3.8 Global, function, and block scope . . . 23
3.9 Variable shadowing . . . 24
3.10 A function that remembers . . . 24
4.1 Creating arrays . . . 27
4.2 Static array methods . . . 27
4.3 Reading and writing by index . . . 28
4.4 Basic array stack and queue methods . . . 28
4.5 slice vs splice . . . 29
4.6 ES2023 Non-Mutating Methods . . . 29
4.7 A tic-tac-toe board as a 2-D array . . . 29
4.8 Destructuring and Spreads . . . 30
5.1 Creating an object literal with dynamic keys . . . 32
5.2 Dot vs Bracket Notation and Optional Chaining . . . 33
5.3 Object Methods and 'this' . . . 33
5.4 Using call(), apply(), and bind() . . . 34
5.5 Reference Mutability and Cloning . . . 34
5.6 Inspecting and Defining Property Descriptors . . . 35
5.7 Creating Objects with Prototypes . . . 36
6.1 Structure of a classic for loop . . . 38
6.2 while vs do...while . . . 39
6.3 for...of (values) vs for...in (keys) . . . 39
6.4 Labeled Loops in Matrix Searches . . . 40
6.5 Generating Infinite Sequences with Generators . . . 41
6.6 Accidental Quadratic Complexity and Linear Optimization . . . 41
7.1 Trap 1: String concatenation . . . 45
7.2 Trap 2: Unintended reference mutation . . . 45
7.3 Trap 3: Truthy empty arrays . . . 45
7.4 Trap 4: Coercion with loose equality . . . 45
7.5 Trap 5: Missing return statement . . . 45
7.6 Trap 6: Block scope bypass . . . 46
7.7 Trap 7: Arrow method this context . . . 46
7.8 Trap 8: Mutating array in loop . . . 46
7.9 Trap 9: Variable shadowing . . . 47

7.10 Trap 10: Floating point math 47
7.11 The Complete Fedi E-Commerce Engine 47
8.1 Passing callbacks to higher-order functions 52
8.2 Transforming arrays with map 53
8.3 Summing and grouping with reduce 53
8.4 Array search utilities 54
8.5 Sorting with Comparison Callbacks 54
8.6 Declarative Data Pipeline 55
9.1 Querying the DOM and Collection Differences 57
9.2 Modifying DOM Content and Styles 58
9.3 High-performance batch insertion with DocumentFragment 59
9.4 Event Bubbling vs Capturing 59
9.5 High-Performance Event Delegation 60
9.6 Implementing Debouncing and Throttling 60
9.7 Creating and Dispatching Custom Events 61
9.8 Viewport Tracking with IntersectionObserver 62
9.9 Subtree Monitoring with MutationObserver 62
10.1 Semantic HTML vs Non-Semantic HTML 65
10.2 ARIA Attributes in Practice 65
10.3 Toggling ARIA State with JavaScript 66
10.4 Focus Management in a Modal Dialog 66
10.5 Announcing Dynamic Updates to Screen Readers 67
10.6 Announcing Form Validation Errors 67
10.7 Building Accessible Forms 68
11.1 Reading a stack trace 71
11.2 Standard Exception Handling 72
11.3 The Finally Return Override Trap 73
11.4 The Async Error Catching Trap 73
11.5 Global Uncaught Rejection Listeners 74
11.6 Custom Application Error Hierarchy 74
11.7 Professional Console Methods 75
11.8 Preventing Detached DOM Memory Leaks 76
12.1 Regex literal vs constructor 78
12.2 Greedy vs Lazy Matching 79
12.3 Regex and String Methods 80
12.4 Named Capture Groups 80
12.5 Using Backreferences 80
12.6 Password Validation with Lookaheads 81
12.7 Example of Catastrophic Backtracking 81
13.1 HTML5 Validation Attributes in Action 84
13.2 Using the Constraint Validation API 85
13.3 Real-Time Inline Validation Controller 86
13.4 Password Confirmation Cross-Validation 87

13.5 Gated Form Submission with fetch() . . . 87
13.6 Safe Output: textContent vs innerHTML . . . 88
14.1 Creating Date Objects . . . 90
14.2 Extracting Date Components . . . 91
14.3 UTC vs Local Time Comparison . . . 91
14.4 Date Arithmetic Utilities . . . 92
14.5 Internationalized Date Formatting . . . 92
14.6 Relative Time Formatting . . . 93
14.7 Building a Countdown Timer . . . 94
15.1 Custom Audio Player HTML Structure . . . 97
15.2 Custom Audio Player JavaScript Controller . . . 97
15.3 Synthesizing a Tone with Web Audio API . . . 99
15.4 Real-Time Frequency Bar Visualizer . . . 99
15.5 Activating Picture-in-Picture Mode . . . 100
15.6 Recording Microphone Audio . . . 100
16.1 Creating and Using Maps . . . 103
16.2 Object Keys are Always Strings . . . 104
16.3 Using Objects and DOM Elements as Map Keys . . . 104
16.4 Map Iteration Patterns . . . 105
16.5 Creating and Using Sets . . . 105
16.6 Set-Theory Operations in JavaScript . . . 106
16.7 WeakMap for Private Data and Caching . . . 106
17.1 Basic localStorage operations . . . 109
17.2 Custom Replacer and Reviver for Complex Storage . . . 110
17.3 Handling Storage Quota Failures Gracefully . . . 111
17.4 A Complete CRUD Task Repository Service . . . 112
17.5 Listening to changes across multiple tabs . . . 113
18.1 Defining and instantiating a class . . . 117
18.2 Enforcing encapsulation with private fields . . . 118
18.3 Using Getters, Setters, and Private Statics . . . 119
18.4 Abstract Classes with new.target . . . 119
18.5 Implementing Mixins in JavaScript . . . 120
18.6 Custom Iterables with Symbol.iterator . . . 121
19.1 A Simple Countdown Using Recursion . . . 124
19.2 Factorial and Power Functions . . . 125
19.3 Sum of Digits and Palindrome Check . . . 125
19.4 Deep Object Key Counter . . . 126
19.5 Recursive Array Flattening . . . 126
19.6 Causing and Preventing Stack Overflow . . . 127
19.7 Tail-Recursive Factorial . . . 127
20.1 Implementing a Stack with modern JS Classes . . . 130
20.2 Implementing an O(1) Queue . . . 131
20.3 Implementing a Doubly Linked List Node and List . . . 132

20.4 Binary Search Tree Implementation . . . 133
20.5 Hash Table with Separate Chaining . . . 134
21.1 Linear Search vs Binary Search . . . 138
21.2 Merge Sort Implementation: O(n log n) . . . 138
21.3 Two Pointers: Target Pair in Sorted Array . . . 140
21.4 Dynamic Sliding Window: Longest Substring without Repeating Characters . . . 140
22.1 Binary Tree Node Class . . . 146
22.2 Recursive Tree Traversal Implementations . . . 146
22.3 BST Insert, Search, and FindMin . . . 147
22.4 Graph Class with Adjacency List . . . 148
22.5 BFS Implementation with Shortest Path Tracking . . . 149
22.6 DFS: Iterative and Recursive Implementations . . . 150
23.1 Fibonacci with Top-Down Memoization . . . 154
23.2 Fibonacci with Bottom-Up Tabulation . . . 154
23.3 Fibonacci with O(1) Space . . . 155
23.4 Climbing Stairs: Three DP Approaches . . . 155
23.5 Coin Change with Bottom-Up Tabulation . . . 156
23.6 Unique Grid Paths with 2D Tabulation . . . 156
23.7 Longest Common Subsequence (LCS) . . . 157
24.1 Pure vs Impure Functions . . . 159
24.2 Immutable Data Patterns in JavaScript . . . 160
24.3 Building a Data Pipeline with Composition . . . 161
24.4 Compose Utility (Right-to-Left Execution) . . . 161
24.5 Currying for Reusable Function Factories . . . 161
24.6 Imperative vs Declarative Data Processing . . . 162
24.7 Closure-Based Counter (No Classes Needed) . . . 163
25.1 Creating and consuming a Promise . . . 166
25.2 Async / Await Syntax . . . 167
25.3 Sending POST Request with fetch() . . . 167
25.4 Implementing Network Timeouts with AbortController . . . 168
25.5 Parallel Execution with Promise Combinators . . . 168
26.1 Greeting Bot HTML Structure . . . 173
26.2 Greeting Bot CSS Stylesheet . . . 174
26.3 Greeting Bot JavaScript Engine . . . 175
27.1 Digital Pet Game HTML Markup . . . 177
27.2 Digital Pet CSS Styles . . . 178
27.3 Digital Pet Simulation Engine . . . 179
28.1 Flashcard Quiz HTML Markup . . . 181
28.2 Flashcard 3D Perspective CSS . . . 182
28.3 Flashcard Quiz JavaScript Engine . . . 183
29.1 Motorcycle Shop Catalog HTML . . . 186
29.2 Catalog Grid CSS . . . 187
29.3 Motorcycle Shop JavaScript Logic . . . 187

30.1 Drum Machine HTML Matrix 190
30.2 Drum Pad Animation Styles 191
30.3 Drum Machine Audio Engine 191
31.1 Weather Planner HTML Dashboard 193
31.2 Weather Dashboard Styles 194
31.3 Weather Planner JavaScript Engine 194
32.1 Expense Tracker HTML Structure 197
32.2 Expense Tracker CSS Styles 198
33.1 Password Generator HTML Layout 201
33.2 Password Generator Styles 202
33.3 Password Generator Logic 203
34.1 Movie Search HTML Layout 205
34.2 Cinema Explorer Styles 205
34.3 Movie Search Engine Logic 206
35.1 Task Manager HTML Structure 209
35.2 Kanban Columns CSS 210
36.1 Fibonacci Visualizer HTML Markup 214
36.2 Fibonacci Complexity Lab Styles 215
37.1 Pathfinder HTML Markup 218
37.2 Pathfinder Matrix CSS 219

PART

Tier 1 — Beginner

TIER 1 — BEGINNER

Welcome to JavaScript. In these seven modules you will learn the core building blocks of every program ever written: values, variables, functions, arrays, objects, and loops. Take your time here — everything that follows is built on this foundation.

1 MODULE

Variables and Strings

Every program you will ever write does the same two things: it *remembers* information, and it *transforms* that information. Variables are how JavaScript remembers. Strings — sequences of text — are the first kind of information we will learn to transform. By the end of this module you will be storing, combining, and reshaping text like a working developer.

LEARNING OBJECTIVES

- Explain what a variable is and how JavaScript stores values
- Declare variables with `let`, `const`, and `var` — and know which to choose
- Apply professional naming conventions
- Create and combine strings with concatenation and template literals
- Use the most common string methods: `toUpperCase`, `slice`, `includes`, and more
- Explain string immutability and why it matters

1.1 What Is a Variable?

Think of your computer's memory as an enormous wall of labeled boxes. A **variable** is a label you stick on one of those boxes so you can find it again. You put a *value* into the box, and whenever you say the label's name, JavaScript fetches the value for you.

```
1 let name = "Fedi";
2 console.log(name);   // JavaScript fetches the value: "Fedi"
```

Code Listing 1.1: Your first variable

Three things happen on line 1:

1. `let` tells JavaScript: "reserve a box."
2. `name` is the label we chose for the box.
3. `= "Fedi"` puts the text value `"Fedi"` into the box. The `=` operator is called the **assignment operator**.

NOTE

The `=` sign in JavaScript does *not* mean "equals" in the mathematical sense. It means "store the right-hand side into the left-hand side." We will meet the real equality operators (`===`) in Module 2.

The value inside a `let` box can be replaced later — that is why it is called a *variable*:

```
1 let score = 10;
```

```
console.log(score);   // 10
score = 25;           // no "let" here: the box already exists
console.log(score);   // 25
```

Code Listing 1.2: Changing a variable's value

Notice the second assignment has no `let`. You declare a variable once; after that you simply use its name. Writing `let score` twice in the same scope is a `SyntaxError` — try it in your console and read the error message. Reading errors calmly is a core developer skill we will train throughout this book.

1.2 `var`, `let`, and `const`

JavaScript has three declaration keywords, and they are not interchangeable.

1.2.1 `const` — the default choice

`const` creates a **constant**: a binding that cannot be reassigned.

```
const birthYear = 2000;
birthYear = 2001;   // TypeError: Assignment to constant variable
```

Code Listing 1.3: Constants cannot be reassigned

This is a feature, not a limitation. When you declare `const`, you make a promise: "this label will always point to this value." That promise makes your code dramatically easier to reason about, and it lets JavaScript engines and your teammates trust the binding.

1.2.2 `let` — when the value must change

Use `let` when you genuinely intend to reassign: counters, accumulators, state that evolves.

```
let currentLevel = 1;
currentLevel = currentLevel + 1;   // level up!
console.log(currentLevel);         // 2
```

Code Listing 1.4: A value that legitimately changes

1.2.3 `var` — the historical keyword

Before 2015 (ES6), `var` was the only option. You will meet it in older tutorials and legacy code, so you must understand it — but you should not use it in new code. `var` ignores block scope (it is *function-scoped*) and allows redeclaration, both of which cause subtle bugs:

```
if (true) {
  var leaked = "I escape the block!";
  let contained = "I stay inside.";
}
console.log(leaked);      // "I escape the block!"  -- surprising
// console.log(contained); // ReferenceError -- as it should be
```

Code Listing 1.5: Why var surprises beginners

BEST PRACTICE

Declare everything with `const` by default. Switch to `let` only when you know the value must change. Never use `var` in new code. Many professional linters (ESLint rule `no-var`) enforce exactly this.

Table 1.1: `var` vs `let` vs `const`

Keyword	Reassignable	Redeclarable	Scope	Use today?
`var`	Yes	Yes	Function	Legacy only
`let`	Yes	No	Block	When reassignment needed
`const`	No	No	Block	Default choice

1.3 Naming Conventions

Names are for humans, not machines. JavaScript enforces a few hard rules, and the community adds style conventions on top.

Hard rules (breaking them is a `SyntaxError`):

- Names may contain letters, digits, _, and $.
- Names must not begin with a digit: `let 2fast` is illegal.
- Reserved words are forbidden: you cannot name a variable `let` or `function`.

Community conventions (breaking them marks you as a beginner):

- Use **camelCase** for variables and functions: `firstName`, `totalPrice`.
- Use **SCREAMING_SNAKE_CASE** for true program-wide constants: `MAX_LOGIN_ATTEMPTS`.
- Names should be pronounceable and searchable: `elapsedTimeInDays`, not `etd`.
- Booleans read best as questions: `isLoggedIn`, `hasPermission`, `canEdit`.

```
const userName = "FediUser";      // camelCase, descriptive
const MAX_RETRIES = 3;            // fixed program constant
let isOnline = true;              // boolean as a question
```

Code Listing 1.6: Names that explain themselves

COMMON MISTAKE

JavaScript names are **case-sensitive**. `userName`, `username`, and `UserName` are three different variables. The classic beginner bug is declaring `let userName` and later reading `userNAme`, producing a confusing `ReferenceError`.

1.4 String Basics

A **string** is a sequence of characters — text. You create one by wrapping characters in single quotes, double quotes, or backticks:

```
const single = 'Hello';
const double = "Hello";
```

```
const backtick = `Hello`;     // a template literal -- more soon
```

Code Listing 1.7: Three ways to write a string

Single and double quotes are interchangeable; pick one style and stay consistent (most professional teams prefer double quotes or backticks, enforced by a formatter). If your text contains an apostrophe, either switch quote styles or escape the character with a backslash:

```
const quote1 = "Fedi's laptop";        // easiest: use double quotes
const quote2 = 'Fedi\'s laptop';       // escaping with backslash
const dialogue = "She said, \"Hi!\""; // escaped double quotes
```

Code Listing 1.8: Quotes inside strings

Every string carries useful built-in information and tools, called **properties** and **methods**. The most fundamental is `length`:

```
const city = "Tunis";
console.log(city.length);   // 5
console.log("".length);     // 0 -- the empty string
```

Code Listing 1.9: The length property

You can reach individual characters with **bracket notation**. JavaScript counts from zero — the first character is at **index 0**:

```
const word = "JavaScript";
console.log(word[0]);    // "J"
console.log(word[4]);    // "S"
console.log(word[word.length - 1]);  // "t" -- last character
console.log(word[99]);   // undefined -- no error, just undefined
```

Code Listing 1.10: Indexing into a string

COMMON MISTAKE

Zero-based indexing feels strange for about a week, then becomes second nature. The last valid index is always `length - 1`. Off-by-one errors (reading `word[length]`) are among the most common bugs in all of programming.

1.5 String Concatenation

Concatenation means joining strings end to end with the + operator:

```
const firstName = "Fedi";
const lastName = "Yahiyaoui";
const fullName = firstName + " " + lastName;
console.log(fullName);   // "Fedi Yahiyaoui"
```

Code Listing 1.11: Joining strings with +

Notice the `"␣"` in the middle — JavaScript never adds spaces for you. Concatenation works, but it gets unreadable fast:

```
const user = "FediUser";
const points = 1280;
const rank = "Gold";

const badge = "Player " + user + " has " + points +
              " points and rank " + rank + ".";
// Easy to misplace a quote or a space. There is a better way...
```

Code Listing 1.12: Concatenation fatigue

1.6 Template Literals

ES6 introduced **template literals**: strings wrapped in backticks that can *embed expressions* directly using `${...}`. This is the modern, professional way to build text.

```
const user = "FediUser";
const points = 1280;
const rank = "Gold";

const badge = `Player ${user} has ${points} points and rank ${rank}.`;
console.log(badge);
// "Player FediUser has 1280 points and rank Gold."
```

Code Listing 1.13: Template literals: readable string building

Anything inside `${}` is a real JavaScript expression — you can compute inline:

```
const price = 49.99;
const quantity = 3;
console.log(`Total: $${(price * quantity).toFixed(2)}`);
// "Total: $149.97"

console.log(`Next year you will be ${2026 - 2000 + 1} years old.`);
```

Code Listing 1.14: Expressions inside placeholders

Template literals also preserve line breaks, making multi-line text natural:

```
const menu = `
=== FediStore ===
1. View products
2. Checkout
3. Exit
`;
console.log(menu);
```

Code Listing 1.15: Multi-line strings

BEST PRACTICE

Prefer template literals over + concatenation for any string that mixes text and values. It is shorter, less error-prone, and instantly readable. Formatters like Prettier will happily convert your code to this style.

1.7 Common String Methods

A **method** is a function attached to a value, called with a dot and parentheses. Strings ship with dozens; these are the ones you will use daily.

1.7.1 Changing case

```
const shout = "javascript is fun";
console.log(shout.toUpperCase());   // "JAVASCRIPT IS FUN"
console.log("LOUD NOISES".toLowerCase());  // "loud noises"
```

Code Listing 1.16: toUpperCase and toLowerCase

Case conversion is the standard trick for comparing user input safely: `answer.toLowerCase() === "yes"` accepts “Yes”, “YES”, and “yes” alike.

1.7.2 Searching inside strings

```
const email = "fedi@fediapp.dev";

console.log(email.includes("@"));        // true
console.log(email.indexOf("@"));         // 4  (index of first match)
console.log(email.indexOf("zzz"));       // -1 (not found)
console.log(email.startsWith("fedi"));   // true
console.log(email.endsWith(".dev"));     // true
```

Code Listing 1.17: includes, indexOf, startsWith, endsWith

NOTE

`includes` returns a boolean (yes/no), while `indexOf` returns a position (or `-1` when absent). Reach for `includes` first — it expresses intent more clearly.

1.7.3 Extracting pieces: `slice`

`slice(start, end)` copies out a section of the string, from `start` up to — but not including — `end`:

```
const lang = "JavaScript";
console.log(lang.slice(0, 4));    // "Java"
console.log(lang.slice(4));       // "Script"  (to the end)
console.log(lang.slice(-6));      // "Script"  (negative = from the end)
```

Code Listing 1.18: Slicing strings

A classic professional pattern — splitting an email into user and domain:

```
const email = "fedi@fediapp.dev";
const at = email.indexOf("@");

const user = email.slice(0, at);        // "fedi"
const domain = email.slice(at + 1);     // "fediapp.dev"
console.log(`user=${user}, domain=${domain}`);
```

Code Listing 1.19: Combining indexOf and slice

1.7.4 Cleaning and replacing

```
const messy = "   hello world   ";
console.log(messy.trim());                // "hello world"

const sentence = "I like cats and cats like me";
console.log(sentence.replace("cats", "dogs"));
// "I like dogs and cats like me"   -- only the FIRST match
console.log(sentence.replaceAll("cats", "dogs"));
// "I like dogs and dogs like me"

console.log("JS ".repeat(3));   // "JS JS JS "
```

Code Listing 1.20: trim, replace, replaceAll, repeat

1.7.5 Splitting strings into arrays

`split` cuts a string into pieces and returns an *array* (Module 4) — a preview of how the language's pieces interlock:

```
const tags = "javascript,web,frontend,fedi";
const tagList = tags.split(",");
console.log(tagList);          // ["javascript", "web", "frontend", "fedi"]
console.log(tagList.length);   // 4

const letters = "abc".split("");
console.log(letters);          // ["a", "b", "c"]
```

Code Listing 1.21: split: from string to array

1.8 String Immutability

Here is a fact that surprises almost every beginner: **strings in JavaScript cannot be changed**. Ever. They are *immutable*.

```
let word = "hello";
word[0] = "H";            // silently does nothing (not an error!)
console.log(word);        // still "hello"

word = word.toUpperCase(); // creates a NEW string, reassigns the label
```

```
console.log(word);          // "HELLO"
```

Code Listing 1.22: Strings never change in place

When you call `word.toUpperCase()`, JavaScript does not modify "hello" — it builds a brand-new string "HELLO" and hands it back to you. If you do not store or use the result, it vanishes. This is why so many string bugs look like "my method call did nothing."

COMMON MISTAKE

The single most common string mistake: calling a method without using its return value. `name.trim();` alone changes nothing. You want `name = name.trim();`.

MINI CHALLENGE

The Username Generator. FediStore needs short usernames. Given `const fullName = "Fedi Yahiyaoui";` and `const birthYear = 2000;`, produce the string `"FYahiyaoui_2000"` using only indexing, `slice`, `indexOf`, and a template literal. Hint: the initial is at index 0, and the surname starts one character after the space.

1.9 Exercises

E1. [Level 1] Declare a constant `favoriteLanguage` with the value `"JavaScript"` and log it. Then try to reassign it and read the error.

E2. [Level 1] Store your city in a variable and log its length and its first and last characters.

E3. [Level 2] Given `const file = "report-final.pdf";`, use string methods to log `true` if the file ends with `".pdf"`, and to extract the name without extension (`"report-final"`).

E4. [Level 2] Build this exact output with a template literal: `FediUser has 3 unread messages.` The name and count must come from variables, and the count must be computed as `1 + 2` inside the placeholder.

E5. [Level 3] Write code that takes `"  FEDI yahiyaoui  "` and produces `"Fedi Yahiyaoui"` — trimmed, with only the first letter of each word capitalized. Combine `trim`, `toLowerCase`, `slice`, `toUpperCase`, indexing, and `indexOf`.

E6. [Level 4] A password is "acceptable" at FediApp when it is at least 8 characters long and contains the symbol `@` or `#`. Given `const password = "fedi@2026";`, build an expression that evaluates to `true` or `false` accordingly, and log a friendly verdict with a template literal. (You will formalize this with `if` statements soon — for now, compose booleans directly.)

1.10 Quiz

Q1. Which keyword should be your default for declarations, and why?

Q2. What is the value of `"JavaScript".indexOf("S")`? And of `"JavaScript"[4]`?

Q3. What does `"abc".toUpperCase()` do to the original string?

Q4. What is printed: `console.log("5"+ 3);` and `console.log("5"- 3);`? (Investigate in your console — the answer reveals how + treats strings.)

Q5. Why is `let 1stPlace` invalid, and what would you name it instead?

Q6. Rewrite with a template literal: `"Hello, "+ name + "! You are level "+ level + "."`

CHAPTER SUMMARY

- Variables are named labels for values in memory; declare with `const` by default, `let` when reassignment is needed, never `var`.
- Use camelCase names; booleans as questions; constants in SCREAMING_SNAKE_CASE.
- Strings are text in quotes or backticks; indexing starts at 0; `length` gives the character count.
- Template literals (`` `...${expr}...` ``) are the professional way to build dynamic text.
- Master the daily methods: `toUpperCase`, `toLowerCase`, `includes`, `indexOf`, `slice`, `trim`, `replace(All)`, `split`, `repeat`.
- Strings are immutable — every method returns a *new* string.

INTERVIEW CORNER

Q: What is the difference between `var`, `let`, and `const`? — Interviewers want: scoping (function vs block), reassignment rules, hoisting behavior, and your personal default (`const` first). Answer with the table from this module and you will sound like you write modern code — because you do.

Q: Are strings mutable in JavaScript? — No. Methods return new strings. Mention that this makes string operations safe to share across a program.

2 MODULE

Booleans and Numbers

Programs make decisions. Should this user see the dashboard? Is the cart empty? Did the player win? Every decision reduces to a value that is either *true* or *false* — a **boolean**. And behind nearly every feature sits arithmetic: prices, scores, timers, coordinates. This module gives you both.

LEARNING OBJECTIVES

- Use boolean values and the three logical operators
- Predict truthiness: which values JavaScript treats as false
- Compare values correctly with === (and avoid ==)
- Perform arithmetic and use the Math object
- Handle floating-point precision and rounding like a professional
- Convert between strings and numbers safely

2.1 Boolean Values

A boolean has exactly two possible values, named after the logician George Boole:

```
const isLoggedIn = true;
const hasPaid = false;

console.log(typeof isLoggedIn);   // "boolean"
```

Code Listing 2.1: The two boolean values

Booleans rarely appear as literals; they are usually *produced* by questions you ask about other values. The moment you write `age >= 18`, you have created a boolean.

2.2 Comparison Operators

Comparisons ask a question about two values and answer with a boolean.

```
const age = 21;

console.log(age > 18);     // true   -- strictly greater
console.log(age >= 21);    // true   -- greater or equal
console.log(age < 18);     // false
console.log(age === 21);   // true   -- strict equality
```

```
console.log(age !== 21);   // false  -- strict inequality
```

Code Listing 2.2: The comparison operators

2.2.1 Always use strict equality

JavaScript has two equality operators, and the difference matters enormously:

```
console.log(5 === "5");    // false -- different TYPES: number vs string
console.log(5 == "5");     // true  -- loose equality CONVERTS types first

console.log(0 == "");      // true  -- (!) empty string coerced to 0
console.log(null == undefined);  // true -- a special historical rule
console.log(0 === "");     // false -- strict: no conversion, no surprises
```

Code Listing 2.3: === vs ==

The loose == performs **type coercion**: it silently converts values to make them comparable, following rules so arcane that even senior developers cannot recite them. === (strict equality) first checks the *type*, then the value. No conversion, no surprises.

COMMON MISTAKE

Never use == or != in new code. Use === and !==. This is not a style preference; it is a bug-prevention rule enforced by virtually every professional style guide (Airbnb, Google, StandardJS).

BEST PRACTICE

One sanctioned exception: `value == null` checks for both `null` and `undefined` in a single test. Some teams allow it. Until you can explain *why* it works, stick to ===.

2.3 Logical Operators

Three operators combine booleans into bigger questions.

```
const age = 25;
const hasLicense = true;

// AND (&&): true only if BOTH sides are true
console.log(age >= 18 && hasLicense);        // true

// OR (||): true if AT LEAST ONE side is true
const isMember = false;
const hasCoupon = true;
console.log(isMember || hasCoupon);          // true

// NOT (!): flips the boolean
console.log(!isMember);                      // true
```

Code Listing 2.4: AND, OR, NOT

A real-world pattern from FediApp — route guarding:

```
const user = { name: "FediUser", role: "admin", isActive: true };

const canAccessDashboard =
  user.isActive && (user.role === "admin" || user.role === "editor");

console.log(canAccessDashboard);   // true
```

Code Listing 2.5: Combining conditions like a professional

Parentheses are not just for grouping — they are documentation. Read that expression aloud: "active, *and* either admin or editor."

Table 2.1: Truth table for `&&` and `||`

a	b	a && b	a || b	!a
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

2.3.1 Short-circuit evaluation

JavaScript is lazy in a useful way: it stops evaluating as soon as the answer is known. For `&&`, if the left side is false, the right side is never touched. For `||`, if the left side is true, the right side is skipped.

```
const user = null;

// Without &&, this would crash: cannot read .name of null
const displayName = user && user.name;
console.log(displayName);   // null -- safe, no error

// || supplies defaults (older style; see ?? below)
const inputName = "";
const name = inputName || "Guest";
console.log(name);   // "Guest"
```

Code Listing 2.6: Short-circuiting in practice

NOTE

Modern JavaScript offers the **nullish coalescing** operator `??`, which only falls back on `null` or `undefined` — not on `""` or `0`: `const name = inputName ?? "Guest";` keeps an intentional empty string. Prefer `??` when `0` or `""` are legitimate values.

2.4 Truthy and Falsy

Every JavaScript value, when placed in a boolean context (like an `if` test), is treated as either **truthy** or **falsy**. There are exactly *eight* falsy values — memorize this list:

Falsy value	Meaning
`false`	the boolean itself
`0`, `-0`	zero (and negative zero)
`0n`	BigInt zero
`""`	the empty string
`null`	deliberate absence
`undefined`	not yet assigned
`NaN`	failed numeric operation

Everything else is truthy — including the surprises: `"0"`, `"false"`, empty arrays `[]`, and empty objects `{}`.

```
console.log(Boolean(""));        // false
console.log(Boolean("0"));       // true  -- non-empty string!
console.log(Boolean([]));        // true  -- empty array is truthy!
console.log(Boolean(0));         // false
console.log(Boolean("Fedi"));    // true
```

Code Listing 2.7: Testing truthiness

COMMON MISTAKE

The classic bug: checking `if (array)` to see whether an array has elements. An *empty* array is truthy! The correct test is `array.length > 0`. Similarly, `"0"` is truthy — a form field containing the text "0" passes a naive emptiness check.

2.5 Number Basics

JavaScript has one number type for everything — integers, decimals, negatives, huge values:

```
const price = 19.99;
const taxRate = 0.07;

console.log(price * 2);          // 39.98  multiplication
console.log(price * taxRate);    // 1.3993...
console.log(10 + 5);             // 15
console.log(10 / 4);             // 2.5  -- real division, no integer truncation
console.log(10 % 3);             // 1   -- remainder (modulo)
console.log(2 ** 10);            // 1024 -- exponentiation
```

Code Listing 2.8: Arithmetic operators

The **remainder operator** `%` is a quiet superpower. `n % 2 === 0` tests evenness; `i % 3 === 0` fires every third iteration; `minutes % 60` wraps a clock. You will use it constantly.

```
const minute = 127;
console.log(`${Math.floor(minute / 60)}h ${minute % 60}m`);  // "2h 7m"

const isEven = 128 % 2 === 0;   // true
```

Code Listing 2.9: Modulo in the wild

2.5.1 Compound assignment and increment

```
let cart = 0;
cart += 3;      // cart = cart + 3   -> 3
cart -= 1;      // -> 2
cart *= 10;     // -> 20
cart++;         // -> 21  (increment by 1)
cart--;         // -> 20  (decrement by 1)
```

Code Listing 2.10: Updating numbers concisely

2.5.2 Special numeric values

```
console.log(1 / 0);          // Infinity  -- not an error!
console.log("abc" * 2);      // NaN -- "Not a Number"
console.log(NaN === NaN);    // false -- NaN equals nothing, even itself
console.log(Number.isNaN(NaN));      // true  -- the correct test
console.log(Number.isFinite(42));    // true
```

Code Listing 2.11: Infinity and NaN

`NaN` signals a numeric operation that produced no meaningful number. Because `NaN === NaN` is `false`, always test with `Number.isNaN()`.

2.6 The Math Object

JavaScript's built-in `Math` object provides constants and functions:

```
console.log(Math.round(4.6));    // 5   -- nearest integer
console.log(Math.floor(4.9));    // 4   -- always rounds DOWN
console.log(Math.ceil(4.1));     // 5   -- always rounds UP
console.log(Math.trunc(4.9));    // 4   -- drops the decimals

console.log(Math.max(3, 9, 5));  // 9
console.log(Math.min(3, 9, 5));  // 3
console.log(Math.abs(-12));      // 12
console.log(Math.sqrt(144));     // 12
console.log(Math.PI);            // 3.14159...
```

Code Listing 2.12: Everyday Math methods

2.6.1 Random numbers

`Math.random()` returns a decimal in $[0, 1)$ — zero inclusive, one exclusive. Every random-integer recipe in JavaScript is built from it:

```
// Random integer from 1 to 6 (a die roll):
const die = Math.floor(Math.random() * 6) + 1;

// General recipe: integer in [min, max] inclusive:
function randomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
```

```
}

console.log(randomInt(1, 100));   // e.g. 47
```

Code Listing 2.13: The standard random-integer recipe

Study that recipe until you can reconstruct it: scale the range (`max - min + 1`), floor it, shift by `min`.

COMMON MISTAKE

`Math.random()` is *not* cryptographically secure. It is fine for games and UI, but never for passwords, tokens, or anything security-related — use `crypto.getRandomValues()` there (Module on form validation touches this again).

2.7 Precision and Rounding

Here is the most famous gotcha in JavaScript:

```
console.log(0.1 + 0.2);                 // 0.30000000000000004
console.log(0.1 + 0.2 === 0.3);         // false  (!)
```

Code Listing 2.14: Floating-point reality

This is not a JavaScript flaw — it is how virtually all computers represent decimals (IEEE 754 binary floating-point). Some decimal fractions simply have no exact binary representation, like $1/3$ has no exact decimal one.

Professional strategies:

```
// Strategy 1: round for display with toFixed (returns a STRING)
const total = 0.1 + 0.2;
console.log(total.toFixed(2));          // "0.30"

// Strategy 2: compare with a small tolerance (epsilon)
const close = Math.abs(total - 0.3) < Number.EPSILON * 10;
console.log(close);                     // true

// Strategy 3: money as integer cents -- the industry standard
const priceCents = 1999;                // $19.99 stored as cents
const taxCents = Math.round(priceCents * 0.07);
console.log(`Total: $${(priceCents + taxCents) / 100}`);  // "Total: $21.39"
```

Code Listing 2.15: Working around float imprecision

BEST PRACTICE

Never store money as floating-point dollars in serious applications. Store integer cents and divide only for display. Every payment system you have ever used works this way.

2.8 Type Conversion

Values flow between strings and numbers constantly — especially from form inputs, which *always* arrive as strings.

```
const input = "42";

console.log(Number(input));       // 42   -- strict, clean
console.log(parseInt("42px"));    // 42   -- parses leading integer
console.log(parseFloat("3.14abc"));// 3.14
console.log(+"42");               // 42   -- unary plus, concise

console.log(Number("abc"));       // NaN  -- failure you can detect
console.log(Number(""));          // 0    -- (!) empty string becomes 0
```

Code Listing 2.16: String to number: the reliable ways

```
const score = 1500;
console.log(String(score));          // "1500"
console.log(score.toString());       // "1500"
console.log(`${score}`);             // "1500" -- template literal
console.log((255).toString(16));     // "ff"  -- radix conversions!
```

Code Listing 2.17: Number to string

And the implicit conversions that cause bugs:

```
console.log("5" + 3);    // "53"   -- + with a string CONCATENATES
console.log("5" - 3);    // 2      -- - always means math
console.log("5" * "2");  // 10
console.log(true + 1);   // 2      -- true becomes 1
```

Code Listing 2.18: Coercion traps

COMMON MISTAKE

The + operator is overloaded: with two numbers it adds; with any string present it concatenates. "My total is wrong" bugs are very often a form value (arriving as a string) being "added" with +. Convert first: `Number(input) + 3`.

MINI CHALLENGE

The FediStore Checkout. A cart contains items priced (in cents) at `2499`, `899`, and `12999`. Compute the subtotal in cents, apply a 7% tax rounded to the nearest cent, and log exactly: `Subtotal: $163.97 | Tax: $11.48 | Total: $175.45` — using integer-cent arithmetic and a template literal. (If your numbers differ, check where you introduced floats.)

2.9 Exercises

E1. [Level 1] Log the truthiness of these with `Boolean()`: `"hello"`, `0`, `"0"`, `[]`, `null`, `-1`. Predict each result *before* running.

E2. [Level 1] Write expressions that are `true` when: (a) `age` is between 13 and 19 inclusive; (b) `name` is not empty; (c) `year` is divisible by 4.

E3. [Level 2] A user typed `"12.5"` into a form. Convert it to a number, multiply by 4, and log the result with two decimals, e.g. `Result: 50.00`.

E4. [Level 2] Simulate two dice with `randomInt(1, 6)` and log whether the roll was a double (both equal).

E5. [Level 3] Without running it, predict the value of each line, then verify: `"10" + 5`, `"10" - 5`, `"" + 1`, `true + true`, `Number("12px")`, `parseInt("12px")`.

E6. [Level 4] FediApp grants free shipping when the cart is at least $50 *or* the user is a member — but never for orders under $5. Given `cartCents`, `isMember`, write a single boolean expression for `freeShipping`, then test it with four scenarios and log a table of results.

2.10 Quiz

Q1. List all eight falsy values from memory.

Q2. Why does `0.1 + 0.2 !== 0.3`? Name two professional workarounds.

Q3. What is the difference between `Number("42")` and `parseInt("42px")`?

Q4. What does `"abc" * 2` evaluate to, and how do you test for that value?

Q5. Explain short-circuiting: why is `user && user.name` safe when `user` is `null`?

Q6. Reconstruct from memory the recipe for a random integer in $[\text{min}, \text{max}]$.

CHAPTER SUMMARY

- Booleans (`true`/`false`) are produced by comparisons; always use strict `===` and `!==`.
- `&&`, `||`, and `!` combine conditions; `&&` and `||` short-circuit, enabling safe property access and defaults (prefer `??` for defaults).
- Eight falsy values: `false`, `0`, `-0`, `0n`, `""`, `null`, `undefined`, `NaN`. Everything else — including `"0"` and `[]` — is truthy.
- `%` (remainder) powers even/odd tests and wrapping arithmetic; `**` exponentiates.
- Floats are imprecise: round for display, compare with tolerance, store money as integer cents.
- Convert input with `Number()`; beware `+` concatenating strings.

INTERVIEW CORNER

Q: What is the difference between == and ===? — Coercion. Give one concrete trap (`0 == ""`) and state your rule: strict only.

Q: What are the falsy values? — Recite the eight. Bonus: mention that empty arrays and objects are truthy, which trips up many candidates.

Q: How do you generate a random integer between 1 and 10? — `Math.floor(Math.random() * 10) + 1`. Explain each step.

3 MODULE

Functions

A function is a named, reusable recipe. You define it once, then execute it as many times as you like, with different ingredients. Functions are the single most important idea in programming: they let you build big programs out of small, testable, nameable pieces. Everything in the rest of this book — objects, callbacks, async code — is functions wearing different clothes.

LEARNING OBJECTIVES

- Declare and call functions with parameters and return values
- Distinguish function declarations, expressions, and arrow functions
- Use default parameters to make functions robust
- Reason about scope: global, function, and block
- Take a first look at closures — functions that remember

3.1 Declaring Functions

```
function greet(name) {
  return `Hello, ${name}! Welcome to FediApp.`;
}

const message = greet("Fedi");     // calling (invoking) the function
console.log(message);              // "Hello, Fedi! Welcome to FediApp."
console.log(greet("Amina"));       // reuse with a different argument
```

Code Listing 3.1: A function declaration

Anatomy of the declaration:

- `function` — the keyword.
- `greet` — the function's name (camelCase, usually a verb).
- `(name)` — a **parameter**: a placeholder variable.
- `{ ... }` — the **body**: the recipe's steps.
- `return` — hands a value back to the caller and *exits* the function.

When you *call* `greet("Fedi")`, the string `"Fedi"` is the **argument** — the actual value poured into the parameter.

NOTE

Terminology: **parameters** live in the definition (the recipe); **arguments** live in the call (the actual ingredients). Developers say "pass an argument to a parameter."

3.2 Return Values

A function's `return` is its answer. Without it, a function performs actions but hands back `undefined`:

```
function add(a, b) {
  return a + b;
}
function addAndLog(a, b) {
  console.log(a + b);   // prints, but returns undefined
}

const sum1 = add(2, 3);          // 5  -- usable
const sum2 = addAndLog(2, 3);    // prints 5, but sum2 is undefined
```

Code Listing 3.2: With and without return

Two rules about `return`:

1. **It exits immediately.** Code after `return` never runs — a pattern called an *early return* or *guard clause*:

```
function getDiscount(user) {
  if (!user) return 0;                    // guard: no user, no discount
  if (!user.isMember) return 0;           // guard: non-members pay full price
  return user.years > 2 ? 0.15 : 0.05;    // happy path at the bottom
}
```

Code Listing 3.3: Guard clauses: fail fast, stay flat

2. **It returns exactly one value.** To give back several, pack them in an array or object (Modules 4–5): `return { min, max };`

COMMON MISTAKE

Putting a line break directly after `return` is a famous trap: JavaScript inserts a semicolon (*ASI*) and your function returns `undefined`. Keep the value on the same line, or wrap multi-line values in parentheses.

3.3 Function Expressions

Functions in JavaScript are *values* — you can store them in variables, pass them around, put them in arrays. A function stored in a variable is a **function expression**:

```
const square = function(n) {
  return n * n;
};

console.log(square(7));        // 49
console.log(typeof square);    // "function" -- a value like any other
```

Code Listing 3.4: A function as a value

One practical difference from declarations: **hoisting**. Function *declarations* are hoisted fully — you can call them above their definition. Function *expressions* follow normal variable rules — calling before the assignment throws.

```
console.log(declared(2));   // 4  -- works: declaration is hoisted
// console.log(expressed(2)); // ReferenceError: cannot access before init

function declared(n) { return n * 2; }
const expressed = function(n) { return n * 2; };
```

Code Listing 3.5: Hoisting in action

3.4 Arrow Functions

ES6 gave us a compact syntax that dominates modern code:

```
// Full form
const add = (a, b) => {
  return a + b;
};

// Concise body: single expression => implicit return
const addShort = (a, b) => a + b;

// One parameter: parentheses optional
const double = n => n * 2;

// No parameters: empty parentheses required
const greet = () => "Hello from FediApp!";

// Returning an object? Wrap it in parentheses
const makeUser = name => ({ name, role: "member" });

console.log(addShort(2, 3), double(21), greet(), makeUser("Fedi"));
```

Code Listing 3.6: Arrow function syntax, step by step

COMMON MISTAKE

The classic arrow mistake: writing `n => { n * 2}`. Curly braces mean "function body," and bodies need an explicit `return`. No braces means the expression is returned automatically. Choose one: braces + `return`, *or* no braces.

Arrow functions also behave differently with `this` (they don't bind their own) — a crucial advanced detail we will master in Modules 5 and 18.

BEST PRACTICE

Modern convention: use arrow functions for short callbacks and one-liners; use named function declarations for top-level logic. Whatever you choose, be consistent within a file.

3.5 Default Parameters

Callers won't always supply every argument. Defaults keep functions robust:

```
function createGreeting(name = "Guest", punctuation = "!") {
  return `Welcome, ${name}${punctuation}`;
}

console.log(createGreeting());              // "Welcome, Guest!"
console.log(createGreeting("Fedi"));        // "Welcome, Fedi!"
console.log(createGreeting("Fedi", "..."));// "Welcome, Fedi..."
```

Code Listing 3.7: Default parameters

Defaults kick in exactly when the argument is `undefined` — not for `null`, `0`, or `""`. This precision is a feature: `0` is often a legitimate value that old `|` tricks would wrongly overwrite.

3.6 Scope

Scope answers the question: *where can this name be seen?*

```
const APP_NAME = "FediApp";     // GLOBAL scope: visible everywhere

function demo() {
  const local = "function scope";   // visible only inside demo()
  if (true) {
    const blocky = "block scope";   // visible only inside the if-block
    console.log(APP_NAME, local, blocky);   // all visible here
  }
  // console.log(blocky);  // ReferenceError: out of scope
}
demo();
// console.log(local);     // ReferenceError: out of scope
```

Code Listing 3.8: Global, function, and block scope

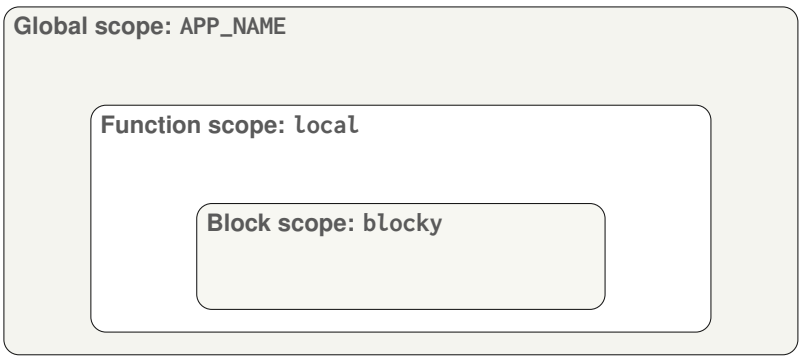


Figure 3.1: Nested scopes: inner scopes can see outward, never inward.

The rule: inner scopes can *see out*; outer scopes cannot *see in*. This protects your variables — each function's internals are private by default.

3.6.1 Shadowing

```
const theme = "dark";

function preview() {
  const theme = "light";    // shadows the outer one inside preview()
  console.log(theme);       // "light"
}
preview();
console.log(theme);         // "dark" -- the global is untouched
```

Code Listing 3.9: Variable shadowing

Shadowing is legal but confusing. Professionals avoid reusing names in nested scopes unless the meaning is genuinely the same.

3.6.2 Global vs local: why globals are dangerous

COMMON MISTAKE

Every global variable is a wire running through your entire program that any line can trip over — or cut. Name collisions, untracked mutations, impossible debugging. Keep variables in the smallest scope possible. Real applications wrap state inside functions, objects, classes, and modules.

3.7 Introduction to Closures

Here is your first taste of one of JavaScript's deepest ideas. When a function is created inside another function, the inner function *remembers* the variables of its birthplace — even after the outer function has finished. That memory is a **closure**.

```
function makeCounter() {
  let count = 0;                  // private: unreachable from outside

  return function() {
    count += 1;                   // the inner function REMEMBERS count
    return count;
  };
}

const counter = makeCounter();
console.log(counter());   // 1
console.log(counter());   // 2
console.log(counter());   // 3
// console.log(count);    // ReferenceError -- truly private!
```

Code Listing 3.10: A function that remembers

`makeCounter` finished long ago, yet `count` lives on — carried inside the returned function. This is how JavaScript achieves *private state* without any special keyword, and it powers callbacks, event handlers, and modules throughout the language.

NOTE

Don't force closures into your mental model all at once. Module 8 (callbacks) and Module 24 (functional programming) will build this muscle gradually. For now, recognize the pattern: *function returning a function that uses outer variables*.

MINI CHALLENGE

The FediTasks Formatter. Write `formatTask(title, done = false)` that returns `"[x] Title"` or `"[ ] Title"`. Then write `makeTodoList()` using a closure: it returns a function that, each call, accepts a title and returns the running list as a numbered string: `1. Buy milk 2. Read Module 3.`

3.8 Exercises

E1. [Level 1] Write `celsiusToFahrenheit(c)` returning `c * 9 / 5 + 32`. Test: 0 → 32, 100 → 212.

E2. [Level 1] Write an arrow function `isAdult` that takes an age and returns a boolean.

E3. [Level 2] Write `repeat(str, times = 2)` that returns the string repeated. `repeat("JS")` → `"JSJS"`.

E4. [Level 2] Write `describeUser(name, role = "member")` returning `"name (role)"`. Call it with one and with two arguments.

E5. [Level 3] Predict before running: what does this print, and why?

```
let x = "outer";
function demo() {
  console.log(x);
  let x = "inner";
}
demo();
```

E6. [Level 4] Write `makeBankAccount(initial)` — a closure returning an object with `deposit(n)`, `withdraw(n)` (refuse overdrafts), and `balance()`. No external code may touch the balance directly.

3.9 Quiz

Q1. Parameter vs argument — define each and place them in a sentence.

Q2. What does a function return when it has no `return` statement?

Q3. Which is hoisted: a function declaration or a function expression?

Q4. When does a default parameter apply — on `null`, `undefined`, or both?

Q5. In your own words: what is a closure?

Q6. Why does `n => { n * 2}` return `undefined`?

CHAPTER SUMMARY

- Functions are reusable recipes: declared once, called many times with different arguments.
- `return` hands back one value and exits; use guard clauses to keep the happy path flat.

- Arrow functions (=>) are the modern concise form; single-expression bodies return implicitly.
- Default parameters guard against `undefined` arguments.
- Scope: inner sees out, outer never sees in; keep variables as local as possible.
- Closures: inner functions remember their birthplace variables — the basis of private state in JavaScript.

INTERVIEW CORNER

Q: Function declaration vs expression vs arrow — differences? — Hoisting (declarations fully hoisted), syntax, and `this` binding (arrows inherit `this`; details in Modules 5 and 18).

Q: What is a closure? — "A function bundled with the variables of the scope where it was created." Give the counter example. This question appears in nearly every JavaScript interview.

4 MODULE

Arrays and Memory Optimization

One variable holds one value. But real programs juggle *collections*: a cart's items, a leaderboard's scores, a feed's posts. An **array** is JavaScript's ordered list — a single name for a numbered sequence of values. Mastering arrays, their methods, and how modern engines represent them in memory is fundamental to writing high-performance applications. In this module, we will explore core array operations, compare mutating and non-mutating APIs, and analyze V8 engine optimization states (Packed vs Holey arrays).

LEARNING OBJECTIVES

- Create, read, and modify arrays using literal syntax and static methods
- Add and remove elements from both ends with `push`, `pop`, `shift`, and `unshift`
- Differentiate between non-destructive `slice` and destructive `splice`
- Iterate arrays with `for`, `for...of`, and `forEach` loops
- Model grids using multidimensional arrays
- Unpack arrays elegantly with destructuring and the spread operator
- Understand V8 engine element kinds: Packed vs Holey optimization transitions
- Apply modern copy-on-write array methods: `toSorted`, `toReversed`, `toSpliced`, and `with`

4.1 Creating Arrays and Static Builders

```
const scores = [85, 92, 78, 90];                    // Array literal
const mixed = ["Fedi", 26, true, null];             // Arrays can hold any type
const empty = [];                                    // The empty array
const slots = new Array(3);                          // [empty x 3] -- creates holes!
const filled = Array.from({ length: 3 }, (_, i) => i + 1);  // [1, 2, 3]
```

Code Listing 4.1: Creating arrays

Always prefer the literal syntax `[...]` over `new Array()`. To check if a variable is an array, use `Array.isArray()` because `typeof []` returns `"object"`:

```
console.log(typeof []);              // "object"
console.log(Array.isArray([]));      // true
console.log(Array.of(10, 20, 30));   // [10, 20, 30]
```

Code Listing 4.2: Static array methods

4.2 Indexing and Length

Like strings, arrays are zero-indexed and know their `length`:

```
const cart = ["keyboard", "mouse", "monitor"];

console.log(cart[0]);              // "keyboard"
console.log(cart[cart.length - 1]); // "monitor" -- the last item
console.log(cart.at(-1));          // "monitor" -- modern negative indexing

cart[1] = "trackpad";              // Arrays are mutable in place!
console.log(cart);                 // ["keyboard", "trackpad", "monitor"]
```

Code Listing 4.3: Reading and writing by index

NOTE

Arrays are references. Declaring an array with `const` prevents reassigning the variable to a new array (`cart = []`), but does **not** prevent mutating the elements inside it (`cart.push("speaker")`).

4.3 V8 Engine Optimization: Packed vs Holey Arrays

Under the hood, modern JavaScript engines like V8 represent arrays in different states based on the types of elements they hold. V8 optimizes arrays that have no gaps (**Packed**) much better than arrays that contain empty slots (**Holey**).

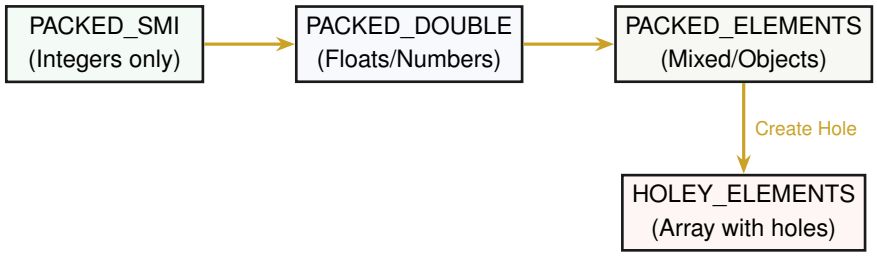


Figure 4.1: V8 Array Elements Optimization Transitions

V8 element kinds transitions are **one-way**: you can transition to a less optimized state, but you can never go back to a more optimized state.

- `PACKED_SMI`: Holds integers only. Fastest representation.
- `PACKED_DOUBLE`: Transitioned when you push a float/double decimal number.
- `PACKED_ELEMENTS`: Transitioned when you add a string, object, or null.
- `HOLEY_ELEMENTS`: Transitioned if you create a gap (e.g. `arr[100] = "x"` on an array of length 5). Once holey, the array remains holey forever, permanently losing fast lookup optimizations.

4.4 Adding and Removing Elements

```
const queue = ["Fedi"];

```

```
queue.push("Amina", "Omar");   // Add at END
console.log(queue);            // ["Fedi", "Amina", "Omar"]

const served = queue.shift();  // Remove from START (slower: shifts indices)
console.log(served);           // "Fedi"

queue.unshift("VIP");          // Add at START (slower)
const last = queue.pop();      // Remove from END (fast)
```

Code Listing 4.4: Basic array stack and queue methods

4.5 Slicing and Splicing

`slice()` extracts a portion of an array and returns a **new copy** (non-destructive). `splice()` deletes or inserts elements **in place** (destructive).

```
const letters = ["a", "b", "c", "d", "e"];

// slice(start, end) -- end is non-inclusive
console.log(letters.slice(1, 3));   // ["b", "c"]
console.log(letters);               // ["a", "b", "c", "d", "e"] (Unchanged!)

// splice(start, deleteCount, ...inserts)
letters.splice(2, 1, "X", "Y");    // Delete 'c', insert 'X', 'Y'
console.log(letters);               // ["a", "b", "X", "Y", "d", "e"] (Mutated!)
```

Code Listing 4.5: slice vs splice

4.6 Modern Non-Mutating Array APIs

ES2023 introduced non-mutating counterparts to copy-on-write arrays:

```
const list = ["git", "node", "vscode"];

const reversed = list.toReversed(); // Returns clone, original untouched
const sorted = list.toSorted();
const replaced = list.with(1, "deno"); // Replaces element at index 1 in clone

console.log(list);     // ["git", "node", "vscode"]
console.log(replaced); // ["git", "deno", "vscode"]
```

Code Listing 4.6: ES2023 Non-Mutating Methods

4.7 Multidimensional Arrays

Arrays can nest other arrays to represent grids, matrices, or board coordinates:

```
const board = [
```

```
  ["X", "O", "X"],
  ["O", "X", "O"],
  ["X", "",  "O"]
];

console.log(board[0][0]);   // "X" -- row 0, col 0
console.log(board[1][2]);   // "O" -- row 1, col 2
```

Code Listing 4.7: A tic-tac-toe board as a 2-D array

4.8 Array Destructuring and Spread Operator

```
const rgb = [247, 223, 30];
const [r, g, b] = rgb; // Destructuring assignment

const [head, ...tail] = [1, 2, 3, 4]; // Rest pattern
console.log(tail); // [2, 3, 4]

// Shallow copying and merging
const base = ["HTML", "CSS"];
const stack = [...base, "JavaScript"]; // ["HTML", "CSS", "JavaScript"]
```

Code Listing 4.8: Destructuring and Spreads

MINI CHALLENGE

The Leaderboard. FediApp shows a top-3 leaderboard from `const scores = [72, 95, 88, 60, 95, 81];`. Without mutating `scores`, produce a new sorted copy (highest first), take the top three elements, and print them as `1) 95  2) 95  3) 88` using `forEach` and destructuring.

4.9 Exercises

E1. [Level 1] Create an array of your three favorite foods. Log the first, the last (using negative indices), and the length.

E2. [Level 1] Start with `[]`, then push `"learn"`, `"build"`, `"repeat"`; pop the last; log the array.

E3. [Level 2] Given `["a","b","c","d","e"]`, produce `["b","c"]` and a full backup copy — original untouched.

E4. [Level 2] Given `["read", "write", "delete"]`, use one `splice` call to replace `"delete"` with `"update"` and insert `"create"` before it. Result: `["read","write","create","update"]`.

E5. [Level 3] Write `sumEven(numbers)` using a `for` loop that totals only the even values. Test with `[1..10]` → 30.

E6. [Level 4] Model a week at FediStore as a 2-D array: 7 rows (days), each `[name, sales]`. Compute the weekly total, find the best day, and print a formatted report with destructuring inside `forEach`.

4.10 Quiz

Q1. Which pair mutates and which copies: `slice/splice`?
Q2. What does `arr[arr.length - 1]` give you, and the modern alternative?
Q3. Why can you `push` into a `const` array?
Q4. `["x","y","z"].splice(1, 0, "w")` — what is the result?
Q5. When do you prefer `for` over `forEach`?
Q6. What is a shallow copy, and why is it dangerous with nested arrays?

CHAPTER SUMMARY

- Arrays are ordered, zero-indexed, mutable collections created with `[...]`.
- `push/pop` work the end; `shift/unshift` the start; all but `length`-reads mutate.
- `slice` copies (safe); `splice` edits in place (powerful, dangerous).
- Iterate with `for` (index, break) or `forEach` (clean visits).
- Nested arrays model grids; peel one bracket at a time.
- Destructure with `[a, b] = arr`; copy and merge with spread `...` — remembering copies are shallow.

INTERVIEW CORNER

Q: Difference between `slice` and `splice`? — The classic. Copy vs mutate; know both signatures cold.

Q: How do you copy an array? — `[...arr]`, `arr.slice()`, and mention shallow vs deep (`structuredClone`) for bonus points.

Q: Is an array passed by value or by reference? — By reference to the same array object; demonstrate with a two-line example.

5 MODULE

Objects and Prototypes

If arrays represent ordered lists, **objects** represent structured entities with named properties. An object is a collection of key-value pairs that model real-world things: a user profile, a database record, an application configuration, or an entire state tree. In JavaScript, almost everything is an object under the hood. Mastering objects, prototypes, property descriptors, and accessor methods is fundamental to writing clean, professional, and secure JavaScript code.

LEARNING OBJECTIVES

- Create objects using object literal syntax and dynamic computed property names
- Access and update properties with dot, bracket, and optional chaining (`?.`) notation
- Understand primitive stack values versus heap object references in memory
- Add methods to objects, handle the `this` keyword, and use `call()`, `apply()`, and `bind()`
- Inspect and control properties with Property Descriptors (`writable`, `enumerable`, `configurable`)
- Implement Getters and Setters for computed object properties
- Protect objects against mutation with `Object.freeze()`, `Object.seal()`, and `Object.preventExtensions()`
- Navigate the Prototype Chain: `Object.create()`, `__proto__`, and prototype inheritance
- Unpack properties using destructuring and clone structures with `structuredClone()`

5.1 Object Literals and Computed Properties

An object literal is a comma-separated list of name-value pairs enclosed in curly braces.

```
const dynamicRoleKey = "user_role";

const fediDeveloper = {
  name: "Fedi",
  role: "Senior Software Engineer",
  experienceYears: 6,
  isAvailableForHire: true,
  skills: ["JavaScript", "Node.js", "React", "Architecture"],
  location: {
    city: "Tunis",
    country: "Tunisia"
  },
  // Computed property name in square brackets:
  [dynamicRoleKey]: "admin"
};

```

```
console.log(fediDeveloper.name); // "Fedi"
console.log(fediDeveloper.user_role); // "admin"
```

Code Listing 5.1: Creating an object literal with dynamic keys

5.2 Accessing Properties: Dot, Bracket, and Optional Chaining

JavaScript provides multiple ways to access object properties. Standard identifiers use dot notation. Dynamic keys, keys with special characters (spaces, hyphens), or variable keys require bracket notation. Optional Chaining (`?.`) prevents runtime errors when navigating deeply nested objects that might be `null` or `undefined`.

```
const book = {
  title: "JavaScript Step by Step",
  "publication-year": 2026,
  author: "Fedi",
  publisher: null
};

// Dot notation: clean, preferred for standard identifiers
console.log(book.title);              // "JavaScript Step by Step"

// Bracket notation: required for hyphens and dynamic variables
console.log(book["publication-year"]); // 2026

const key = "author";
console.log(book[key]);                // "Fedi"

// DANGER: Accessing property of null throws error
// console.log(book.publisher.name);   // TypeError: Cannot read properties of null

// SAFE: Optional chaining returns undefined instead of throwing
console.log(book.publisher?.name);    // undefined
console.log(book.marketing?.budget ?? 0); // 0 (using nullish coalescing)
```

Code Listing 5.2: Dot vs Bracket Notation and Optional Chaining

5.3 Methods, `this`, and Explicit Binding

A method is an object property that holds a function. Inside a method, the `this` keyword refers to the object on which the method was invoked.

```
const userAccount = {
  username: "fediUser",
  balanceCents: 15000,

  deposit(cents) {
    this.balanceCents += cents;
    console.log(`Deposited $${cents / 100}. New balance: $${this.balanceCents / 100}`);
```

```
  },

  getFormattedBalance() {
    return `$${(this.balanceCents / 100).toFixed(2)}`;
  }
};

userAccount.deposit(2500); // New balance: $175
console.log(userAccount.getFormattedBalance()); // "$175.00"
```

Code Listing 5.3: Object Methods and 'this'

COMMON MISTAKE

Arrow functions do **not** bind their own `this`. They inherit `this` lexically from the surrounding scope. Never use arrow functions to define object methods if they need to access the object's properties.

If we extract a method from an object and execute it as a standalone function, the binding to the original object is lost. We can explicitly bind the execution context using `call()`, `apply()`, or `bind()`:

```
const customer = { name: "Fedi" };
function greet(greeting, punctuation) {
  console.log(`${greeting}, ${this.name}${punctuation}`);
}

// 1. call(): invokes function immediately, arguments passed individually
greet.call(customer, "Hello", "!"); // "Hello, Fedi!"

// 2. apply(): invokes function immediately, arguments passed as array
greet.apply(customer, ["Welcome", "."]); // "Welcome, Fedi."

// 3. bind(): returns a NEW function with locked 'this', does not execute immediately
const boundGreet = greet.bind(customer, "Hi");
boundGreet("!"); // "Hi, Fedi!"
```

Code Listing 5.4: Using call(), apply(), and bind()

5.4 Object References and Heap Memory

Unlike primitive values (which are copied by value in stack memory), objects are stored by reference in heap memory. A variable does not contain the object itself; it contains a pointer to the object's address on the heap.

```
const original = { title: "Draft Post", published: false };
const copy = original; // Copies the REFERENCE!
copy.published = true;
console.log(original.published); // true! (Both variables mutated the same heap object)

// Shallow clone with spread syntax (only copies properties one-level deep)
const shallowClone = { ...original };
```

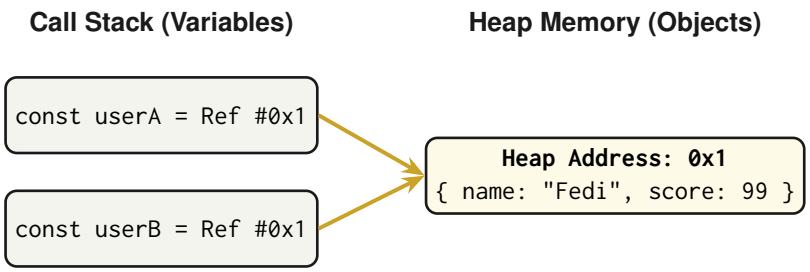


Figure 5.1: Variables Storing Pointers to the Same Object in Heap Memory

```

// Deep clone with modern structuredClone() (safely copies nested objects and arrays)
const deepCopy = structuredClone(original);
```

Code Listing 5.5: Reference Mutability and Cloning

5.5 Property Descriptors and Getters/Setters

Properties can be categorized as **Data Properties** (holding a value) or **Accessor Properties** (using getter/setter functions). You can define and control their behavior using Property Descriptors via `Object.defineProperty()`:

```
const config = {};

Object.defineProperty(config, "API_ENDPOINT", {
  value: "https://api.fediapp.dev/v1",
  writable: false,     // Value cannot be changed
  enumerable: true,    // Appears in loops and Object.keys()
  configurable: false  // Cannot be deleted or reconfigured
});

// Accessor Properties: Getter and Setter
const person = {
  firstName: "Fedi",
  lastName: "Yahiyaoui",
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
  set fullName(value) {
    const parts = value.split(" ");
    this.firstName = parts[0] || "";
    this.lastName = parts[1] || "";
  }
};

console.log(person.fullName); // "Fedi Yahiyaoui"
person.fullName = "Fedi User";
console.log(person.firstName); // "Fedi"
```

Code Listing 5.6: Inspecting and Defining Property Descriptors

5.6 Object Mutation Controls

JavaScript provides three levels of protection against object mutation:

1. **Prevent Extensions** (`Object.preventExtensions(obj)`): Prevents adding new properties. Existing properties can still be modified and deleted.
2. **Seal** (`Object.seal(obj)`): Prevents adding or deleting properties. Marks all existing properties as non-configurable. Existing properties can still be modified if writable.
3. **Freeze** (`Object.freeze(obj)`): Prevents all additions, deletions, and modifications. Marks all properties as non-writable and non-configurable.

5.7 The Prototype Chain

Every object in JavaScript has an internal link to another object called its **prototype**. When you access a property that does not exist on the object itself, JavaScript searches down the **prototype chain** until it finds the property or reaches `null`:

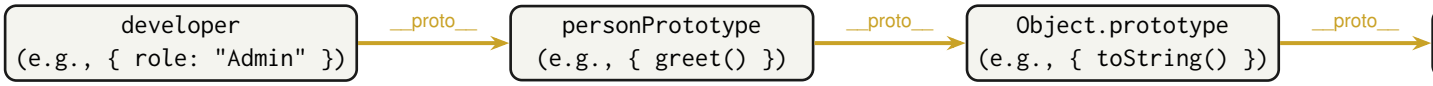


Figure 5.2: The Prototype Inheritance Chain

```
const personProto = {
  greet() {
    return `Hello, I am ${this.name}`;
  }
};

// Create a new object with personProto as its prototype
const dev = Object.create(personProto);
dev.name = "Fedi";
dev.role = "Developer";

console.log(dev.greet()); // "Hello, I am Fedi" (Inherited from personProto!)
console.log(Object.getPrototypeOf(dev) === personProto); // true
```

Code Listing 5.7: Creating Objects with Prototypes

MINI CHALLENGE

The Deep Config Merger. Create a function `mergeConfigs(defaultCfg, userCfg)` that merges a default configuration object with overrides from a user configuration object. Neither input object must be mutated. If a property is itself an object, merge it recursively. Use recursion and standard object iteration methods.

5.8 Exercises

E1. [Level 1] Create a `laptop` object with `brand`, `model`, and `ramGb`. Access properties with dot and bracket notation.

E2. [Level 1] Add a method `getSpecs()` that returns `"Apple M3 Pro with 36GB RAM"` using standard object template syntax.
E3. [Level 2] Write a function `countProperties(obj)` that returns the total number of keys in an object.
E4. [Level 2] Define a non-enumerable, read-only property on an object using `Object.defineProperty()`.
E5. [Level 3] Implement a deep clone function `deepClone(obj)` using `structuredClone()`.
E6. [Level 4] Write a function `diffObjects(objA, objB)` returning keys that were added, modified, or removed, including old and new values.

5.9 Quiz

Q1. When is bracket notation strictly required over dot notation?
Q2. What happens when two variables reference the same heap object and one mutates a property?
Q3. What is the difference between `Object.freeze()` and `Object.seal()`?
Q4. What are the three flags in a Property Descriptor?
Q5. Why is `structuredClone()` preferred over `JSON.parse(JSON.stringify(obj))` for deep cloning?

CHAPTER SUMMARY

- Objects store key-value properties in heap memory; variables store reference pointers.
- Accessor properties utilize getters and setters to run code when reading/writing keys.
- Use Property Descriptors to enforce read-only, non-configurable, or non-enumerable properties.
- Objects inherit properties through the Prototype Chain until resolving to `null`.

INTERVIEW CORNER

Q: How are primitives and objects stored differently in JavaScript? — Primitives (number, string, boolean, etc.) are stored by value in stack memory. Objects (arrays, functions, plain objects) are allocated on the heap, and variables hold reference addresses.

Q: What is prototype pollution? — Prototype pollution is a security vulnerability where an attacker modifies the shared `Object.prototype` (usually via unsafe input parsing in deep merge or JSON parsers). Because all objects inherit from `Object.prototype`, adding properties to it exposes them on all objects in the application, leading to privilege escalation or remote code execution. Avoid this by checking key keys like `__proto__` and `constructor` during deep merges.

6 MODULE

Loops and Iteration

Computers excel at repeating tasks millions of times without fatigue or error. **Loops** are the control structures that command the computer to repeat operations until a specific condition is met. Whether you are processing financial transactions, rendering game frames, searching through datasets, or generating infinite sequences, loops are the workhorses of algorithmic execution. In this module, we will dive deep into classic and modern iteration constructs in JavaScript, analyze performance hazards like quadratic complexity, and master memory-efficient lazy evaluation with generators.

LEARNING OBJECTIVES

- Master the classic three-part `for` loop structure
- Use `while` and `do...while` loops for condition-driven repetition
- Iterate cleanly through collections with modern `for...of` loops
- Inspect object properties with `for...in` loops and avoid prototype pollution hazards
- Control execution flow precisely with `break`, `continue`, and Labeled Statements
- Generate lazy evaluation sequences with Generator Functions (`function*` and `yield`)
- Analyze nested loop performance and avoid accidental quadratic complexity ($O(n^2)$)
- Understand asynchronous iteration with `for await...of` preview

6.1 The Classic `for` Loop

The standard `for` loop is the oldest and most versatile iteration tool in JavaScript. It uses an explicit counter to run a block of code a predetermined number of times.

```
// for (initialization; condition; final-expression)
for (let i = 0; i < 5; i++) {
  console.log(`Iteration number: ${i}`);
}
```

Code Listing 6.1: Structure of a classic for loop

The lifecycle of a `for` loop consists of four distinct phases:

1. **Initialization** (`let i = 0`): Runs exactly once before the loop begins. It declares and initializes the loop counter.
2. **Condition Evaluation** (`i < 5`): Evaluated before every iteration. If `true`, the loop body executes; if `false`, the loop terminates.
3. **Body Execution**: The code block inside the curly braces runs.
4. **Final-expression** (`i++`): Evaluated at the end of each iteration, typically incrementing or decrementing the counter.

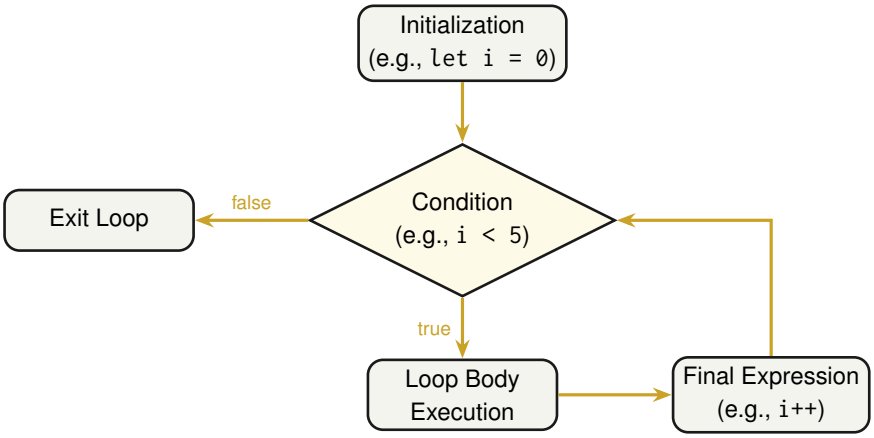


Figure 6.1: Lifecycle and Control Flow of a Classic `for` Loop

6.2 Condition-Driven Iteration: `while` and `do...while`

When the number of iterations is not known in advance, we use condition-driven loops. The `while` loop checks the condition *before* executing the body, whereas the `do...while` loop guarantees the body executes *at least once* because the check occurs at the end.

```
// while: tests condition BEFORE running body
let score = 100;
while (score > 0) {
  score -= 25;
  console.log(`Score: ${score}`);
}

// do...while: GUARANTEES at least one execution
let attempts = 0;
do {
  attempts++;
  console.log(`Running check attempt #${attempts}`);
} while (attempts < 1);
```

Code Listing 6.2: while vs do...while

COMMON MISTAKE

Infinite loops occur when the loop condition never evaluates to `false`. If your program hangs, press `Ctrl + C` in your terminal to force termination, and inspect your loop body to ensure the control variable is correctly updated.

6.3 Modern Iteration: `for...of` vs `for...in`

JavaScript has two distinct “for-in” styles. They serve completely different purposes and confusing them is a major source of bugs.

```
const skills = ["JavaScript", "React", "Node.js"];

```

Table 6.1: for...of vs for...in Comparison

Feature	`for...of`	`for...in`
Iterates Over	**Values** of an iterable	**Keys** (properties) of an object
Applicable to	Arrays, Strings, Maps, Sets	Plain Objects
Respects Order	Yes (guaranteed by iterable)	No (keys are unordered strings)
Includes Prototypes	No	Yes (inherits enumerable properties)
Best For	Processing array elements	Debugging or serializing objects

```
// Add a custom property to the Array prototype (anti-pattern)
Array.prototype.customProperty = "HAZARD";

// for...of iterates over values (safe from prototype pollution)
for (const skill of skills) {
  console.log(`Skill: ${skill}`); // JavaScript, React, Node.js
}

// for...in iterates over keys (includes index and prototype properties!)
for (const index in skills) {
  console.log(`Index: ${index} -> ${skills[index]}`);
  // Logs: "0 -> JavaScript", "1 -> React", "2 -> Node.js", "customProperty -> HAZARD"
}
```

Code Listing 6.3: for...of (values) vs for...in (keys)

BEST PRACTICE

Never use `for...in` to iterate over arrays. It converts indices to strings (e.g., `"0"`), is significantly slower than `for...of` or standard `for` loops, and accesses properties added to the array prototype. Use `for...of` for arrays and collections, and save `for...in` for plain objects.

6.4 Labeled Statements and Controlling Loop Flow

You can skip single iterations with `continue` or abort a loop entirely with `break`. When loops are nested, these operators only affect the innermost loop. To control an outer loop, you must define a **Labeled Statement**.

```
const matrix = [
  [1, 2, 3],
  [4, 99, 6],
  [7, 8, 9]
];

const target = 99;
let foundCoordinates = null;

// 'matrixSearch' is our loop label:
matrixSearch: for (let r = 0; r < matrix.length; r++) {
  for (let c = 0; c < matrix[r].length; c++) {
    if (matrix[r][c] === target) {
      foundCoordinates = { r, c };
```

```
      break matrixSearch; // Instantly halts BOTH outer and inner loops!
    }
  }
}

console.log("Found at:", foundCoordinates); // { r: 1, c: 1 }
```

Code Listing 6.4: Labeled Loops in Matrix Searches

6.5 Generators and Lazy Evaluation

A **Generator Function** (`function*`) yields execution control back to the caller, producing values **on demand (lazily)** rather than computing them all upfront. This is highly memory-efficient when dealing with massive or infinite sequences.

```
// Generates an infinite sequence of ID numbers on demand
function* idGenerator(prefix = "ID") {
  let count = 1;
  while (true) {
    yield `${prefix}-${count++}`;
  }
}

const gen = idGenerator("FEDI");
console.log(gen.next().value); // "FEDI-1"
console.log(gen.next().value); // "FEDI-2"
console.log(gen.next().value); // "FEDI-3"

// Finite ranges using generator loops
function* range(start, end, step = 1) {
  for (let i = start; i <= end; i += step) {
    yield i;
  }
}

for (const num of range(10, 30, 10)) {
  console.log(`Step: ${num}`); // 10, 20, 30
}
```

Code Listing 6.5: Generating Infinite Sequences with Generators

6.6 Nested Loops and Performance Pitfalls

A nested loop runs its inner loop for every single iteration of the outer loop. This leads to **quadratic complexity** ($O(n^2)$). If $n = 100,000$, a nested loop executes $10,000,000,000$ (ten billion) times, locking the thread and freezing your application.

```
const listA = ["fedi", "sam", "alex", "jordan", /* ... 10,000 users */];
const listB = ["alex", "jordan", "taylor", /* ... 10,000 users */];
```

```
3
4  // IMPERATIVE O(n^2): Nesting indexOf/includes inside a loop
5  const commonUsersSlow = [];
6  for (const user of listA) {
7    if (listB.includes(user)) { // includes() is a hidden loop!
8      commonUsersSlow.push(user);
9    }
10 }
11
12 // OPTIMIZED O(n): Convert listB to a Set first
13 const setB = new Set(listB); // O(n) space and time
14 const commonUsersFast = [];
15 for (const user of listA) {
16   if (setB.has(user)) { // Set.has() is O(1) average lookup time!
17     commonUsersFast.push(user);
18   }
19 }
```

Code Listing 6.6: Accidental Quadratic Complexity and Linear Optimization

MINI CHALLENGE

The Spiral Matrix Challenge. Write a function `spiralMatrix(n)` that accepts an integer n and generates an $n \times n$ matrix (array of arrays) filled with numbers from 1 to n^2 in spiral order. For $n = 3$, the output is: `[[1,2,3],[8,9,4],[7,6,5]]`.

6.7 Exercises

E1. [Level 1] Write a `for` loop that prints all even numbers from 2 to 20.
E2. [Level 1] Write a `while` loop halving a number `let n = 1024;` until < 1.
E3. [Level 2] Write a function `sumArray(numbers)` using a `for...of` loop.
E4. [Level 2] Search a 2D matrix for a value using a labeled loop and return coordinates.
E5. [Level 3] Write a generator function `fibonacciGenerator()` yielding infinite Fibonacci numbers lazily.
E6. [Level 4] Write a function `spiralMatrix(n)` generating an $n \times n$ matrix filled in spiral order from 1 to n^2.

6.8 Quiz

Q1. What are the three control expressions in a classic `for` loop?
Q2. How does `do...while` guarantee at least one execution pass?
Q3. What is the purpose of a labeled statement in JavaScript?
Q4. How do generator functions with `yield` save memory over large arrays?
Q5. Why is `for...in` discouraged for iterating arrays?

CHAPTER SUMMARY

- `for` manages indexed counting iterations; `while` handles condition-driven loops.
- `for...of` iterates collection values; `for...in` inspects object keys.

- Labeled statements allow breaking out of multi-level nested loops instantly.
- Generators (`function*`) produce lazy, memory-efficient on-demand sequences via `yield`.

INTERVIEW CORNER

Q: What is the difference between an iterator and a generator in JavaScript? — An iterator is an object implementing the `next()` method returning `{ value, done }`. A generator is a special function (`function*`) that returns a generator iterator and allows pausing and resuming execution using the `yield` keyword.

Q: Why are Set lookups faster than Array lookups? — Arrays store values sequentially, so searching with `includes()` takes $O(n)$ time. Sets are backed by hash tables, allowing lookups with `has()` to run in $O(1)$ average time, regardless of the collection's size.

7 MODULE

JavaScript Fundamentals Review

Congratulations on reaching the end of **Tier 1: Beginner**. In the preceding six modules, you learned the building blocks of every JavaScript application: storing data in variables, evaluating conditions with booleans, organizing logic into functions, manipulating ordered collections with arrays, modeling entities with objects, and automating repetitive tasks with loops.

Before advancing to Tier 2 (Intermediate), where we explore higher-order functions, the browser DOM, asynchronous code, and persistent storage, this module serves as a comprehensive diagnostic review and synthesis of your fundamental knowledge.

LEARNING OBJECTIVES

- Synthesize variables, types, functions, arrays, objects, and control flow
- Identify and fix the 10 most common beginner anti-patterns and bugs
- Trace complex code execution and predict scope and reference outcomes
- Complete the 20-Question Tier 1 Diagnostic Exam
- Build the Tier 1 Synthesis Project: The Fedi Mini E-Commerce Engine

7.1 Core Concepts Architecture

Let us review how all Tier 1 concepts fit into a cohesive mental model:

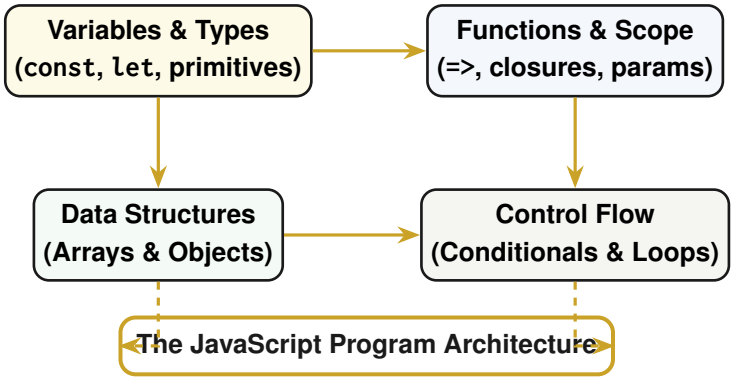


Figure 7.1: The Interlocking Architecture of JavaScript Fundamentals

7.2 The Top 10 Beginner Traps and Anti-Patterns

7.2.1 Trap 1: Accidental String Concatenation with Numbers

```
// WRONG: input from forms is a string!
const quantity = "5";
const total = quantity + 1; // "51" (string concatenation)

// CORRECT: convert explicitly
const totalCorrect = Number(quantity) + 1; // 6
```

Code Listing 7.1: Trap 1: String concatenation

7.2.2 Trap 2: Mutating Objects by Shared Reference

```
// WRONG: shallow reference sharing
const defaultSettings = { theme: "light", notifications: true };
const userSettings = defaultSettings;
userSettings.theme = "dark"; // Mutated defaultSettings as well!

// CORRECT: clone properties
const safeSettings = { ...defaultSettings, theme: "dark" };
```

Code Listing 7.2: Trap 2: Unintended reference mutation

7.2.3 Trap 3: Testing Array Emptiness with Truthiness

```
const results = [];
// WRONG: [] is truthy in JavaScript!
if (results) { /* always executes! */ }

// CORRECT: check length explicitly
if (results.length > 0) { /* only executes if elements exist */ }
```

Code Listing 7.3: Trap 3: Truthy empty arrays

7.2.4 Trap 4: Loose Equality Coercion

```
// WRONG: loose equality causes unexpected matches
if (0 == "") { /* evaluates to true */ }

// CORRECT: always use strict equality
if (0 === "") { /* evaluates to false */ }
```

Code Listing 7.4: Trap 4: Coercion with loose equality

7.2.5 Trap 5: Forgetting to Return from a Function

```
// WRONG: function logs instead of returning
function add(a, b) {
  const result = a + b;
}
const sum = add(4, 5); // sum is undefined!

// CORRECT: return the computed value
function addClean(a, b) {
```

```
  return a + b;
}
```

Code Listing 7.5: Trap 5: Missing return statement

7.2.6 Trap 6: Block Scope vs Function Scope (var hoisting)

```
// WRONG: var ignores curly braces block boundaries!
if (true) {
  var user = "Fedi";
}
console.log(user); // "Fedi" (leaked out of the block!)

// CORRECT: use let/const to enforce block scoping
if (true) {
  const admin = "FediUser";
}
// console.log(admin); // ReferenceError: admin is not defined
```

Code Listing 7.6: Trap 6: Block scope bypass

7.2.7 Trap 7: Lexical "this" in Arrow Methods

```
// WRONG: Arrow function doesn't bind this to current object context!
const profile = {
  name: "Fedi",
  greet: () => {
    return `Hello, I'm ${this.name}`; // this points to outer scope (window/global)
  }
};
console.log(profile.greet()); // "Hello, I'm undefined"

// CORRECT: use standard method syntax
const safeProfile = {
  name: "Fedi",
  greet() {
    return `Hello, I'm ${this.name}`;
  }
};
```

Code Listing 7.7: Trap 7: Arrow method this context

7.2.8 Trap 8: Modifying Array Elements During Iteration

```
// WRONG: changing array length during loop causes skipping!
const numbers = [1, 2, 3, 4, 5];
for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] === 2) {
    numbers.splice(i, 1); // Removes 2, next index skips element 3!
  }
}

// CORRECT: iterate backwards or use filter
```

```
const filtered = numbers.filter(n => n !== 2);
```

Code Listing 7.8: Trap 8: Mutating array in loop

7.2.9 Trap 9: Variable Shadowing Confusion

```
// WRONG: re-declaring variable in inner scope hides outer one
const score = 10;
function updateScore() {
  const score = 20; // Shadows outer variable instead of updating it!
}
updateScore();
console.log(score); // 10 (unchanged!)
```

Code Listing 7.9: Trap 9: Variable shadowing

7.2.10 Trap 10: Floating Point Precision Errors

```
// WRONG: binary representation causes decimal rounding errors
console.log(0.1 + 0.2 === 0.3); // false

// CORRECT: round to specific decimal places or use cents
console.log(Math.round((0.1 + 0.2) * 100) / 100 === 0.3); // true
```

Code Listing 7.10: Trap 10: Floating point math

7.3 Synthesis Project: The Fedi Mini E-Commerce Engine

Let us integrate variables, arrays, objects, functions, and loops into a complete, runnable e-commerce cart calculation engine:

```
// 1. Initial Store Catalog Data
const fediStoreCatalog = [
  { id: "PROD-01", name: "JavaScript Masterclass", priceCents: 4999, category: "education
    ↪ " },
  { id: "PROD-02", name: "Mechanical Keyboard", priceCents: 12900, category: "hardware" },
    ↪
  { id: "PROD-03", name: "Ergonomic Desk Mat", priceCents: 2450, category: "accessories" }
    ↪
];

// 2. Cart State
const shoppingCart = [
  { productId: "PROD-01", quantity: 2 },
  { productId: "PROD-03", quantity: 1 }
];

// 3. Calculation Engine Functions
function findProduct(catalog, productId) {
  for (const item of catalog) {
    if (item.id === productId) {
      return item;
```

```
    }
  }
  return null;
}

function calculateCartTotals(cart, catalog, discountCode = "") {
  let subtotalCents = 0;

  for (const cartItem of cart) {
    const product = findProduct(catalog, cartItem.productId);
    if (product) {
      subtotalCents += product.priceCents * cartItem.quantity;
    }
  }

  // Apply coupon logic
  let discountMultiplier = 0;
  if (discountCode === "FEDI20") {
    discountMultiplier = 0.20; // 20% off
  }

  const discountCents = Math.round(subtotalCents * discountMultiplier);
  const discountedSubtotal = subtotalCents - discountCents;
  const taxCents = Math.round(discountedSubtotal * 0.08); // 8% sales tax
  const finalTotalCents = discountedSubtotal + taxCents;

  return {
    subtotal: `$${(subtotalCents / 100).toFixed(2)}`,
    discount: `$${(discountCents / 100).toFixed(2)}`,
    tax: `$${(taxCents / 100).toFixed(2)}`,
    total: `$${(finalTotalCents / 100).toFixed(2)}`
  };
}

// 4. Execution and Logging
const receipt = calculateCartTotals(shoppingCart, fediStoreCatalog, "FEDI20");
console.log("=== FediStore Receipt ===");
console.log(`Subtotal: ${receipt.subtotal}`);
console.log(`Discount: ${receipt.discount}`);
console.log(`Tax (8%): ${receipt.tax}`);
console.log(`Final:    ${receipt.total}`);
```

Code Listing 7.11: The Complete Fedi E-Commerce Engine

7.4 Tier 1 Diagnostic Self-Check Exam

Test your mastery with these 10 diagnostic questions:

Q1. What is the output of `typeof null` and why? (Answer: `"object"` due to a legacy JS bug).

Q2. What does `const x = (10, 20, 30);` assign to x? (Answer: `30`, the comma operator evaluates to the last expression).

Q3. Predict the output of `[1, 2, 3] + [4, 5, 6]`. (Answer: `"1,2,34,5,6"` via string coercion).

Q4. Does `const obj = { a: 1}; obj.a = 2;` throw an error? (Answer: No, `const` prevents variable reassignment, not object property mutation).

Q5. What is the value of `0 || "Fedi" && false`? (Answer: `false` because `&&` has higher precedence than `||`).

Q6. Why does `Number.isNaN("abc")` return `false` while `isNaN("abc")` returns `true`? (Answer: `isNaN` coerces to number first, `Number.isNaN` strictly checks type).

Q7. How can you swap two variables `x` and `y` in one line using ES6? (Answer: `[x, y] = [y, x]`).

Q8. What is the result of `["a", "b", "c"].slice(1, -1)`? (Answer: `["b"]`).

Q9. What is the difference between lexical scope and dynamic scope? (Answer: Lexical scope is determined at write time by code position; dynamic scope is determined at runtime by call site).

Q10. What will `console.log(1< 2< 3)` and `console.log(3> 2> 1)` output? (Answer: `true` because `1 < 2` is `true`, `true < 3` is `1 < 3 -> true`; but `3 > 2` is `true`, `true > 1` is `1 > 1 -> false`).

CHAPTER SUMMARY

- Tier 1 established your foundation: primitive types, reference types, scope hierarchies, and loops.
- Always enforce strict equality (`===`), explicit type conversion (`Number()`), and immutability practices.
- You are now fully prepared to enter **Tier 2: Intermediate**, where we bridge pure JavaScript with real-world browser interfaces and APIs.

PART II

Tier 2 — Intermediate

TIER 2 — INTERMEDIATE

You know the syntax. Now you learn to think like a developer: higher-order functions, the DOM, events, accessibility, debugging, regular expressions, forms, dates, media, Maps and Sets, and persistent state with localStorage.

8 MODULE

Higher Order Functions and Callbacks

In Tier 1, we processed data using imperative `for` and `while` loops. In Tier 2, we elevate our programming style to **declarative data pipelines**. Instead of instructing the computer *how* to step through each index, we declare *what* transformation we want using **Higher-Order Functions (HOFs)**. HOFs are functions that accept other functions as arguments or return functions as outputs. This module covers first-class functions, callbacks, standard transformations, method chaining, and modern non-mutating array APIs.

LEARNING OBJECTIVES

- Understand functions as first-class citizens and the mechanics of callbacks
- Differentiate between synchronous callbacks and asynchronous callbacks
- Transform array elements declaratively with `map()`
- Extract subsets of data matching criteria with `filter()`
- Fold arrays into single values or structures using `reduce()`, and handle initial value traps
- Search collections using `find()`, `findIndex()`, `some()`, and `every()`
- Sort arrays safely with comparison callbacks in `sort()` and modern `toSorted()`
- Chain array methods into clean, readable data processing pipelines

8.1 First-Class Functions and Callbacks

In JavaScript, functions are **first-class citizens**: they are treated as values. You can assign them to variables, store them in objects or arrays, pass them as arguments to other functions, and return them from functions. When a function is passed as an argument to be called later, it is referred to as a **callback**.

```
// Higher-Order Function that accepts a callback 'action'
function executeWithLogging(label, action) {
  console.log(`[START] ${label}`);
  action(); // Invoking the passed callback function
  console.log(`[END] ${label}`);
}

// Passing an anonymous arrow function as callback
executeWithLogging("Database Backup", () => {
  console.log("Backing up 10,000 user records for FediApp...");
});
```

Code Listing 8.1: Passing callbacks to higher-order functions

8.2 Array Transformations: Map, Filter, and Reduce

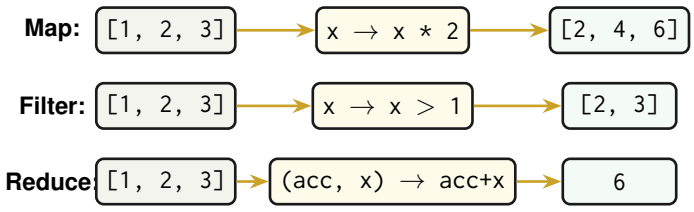


Figure 8.1: Visualizing Map, Filter, and Reduce Operations

Transforms each element individually. Returns a new array of the same length.
Evaluates a truthy/falsy condition for each element. Returns a new array containing only elements that pass the test.
Folds all elements into a single accumulator value (like a sum, an object, or a flat array).

```
const products = [
  { id: 1, name: "Laptop", price: 1000 },
  { id: 2, name: "Mouse", price: 50 },
  { id: 3, name: "Keyboard", price: 120 }
];

const productNames = products.map(item => item.name);
console.log(productNames); // ["Laptop", "Mouse", "Keyboard"]
```

Code Listing 8.2: Transforming arrays with map

8.3 Aggregating with `reduce()`: The Initial Value Trap

The second argument to `reduce()` is the initial value of the accumulator. If omitted:

map(filter() The accumulator defaults to the first element of the array (index 0).

2. Iteration begins at the second element (index 1) instead of index 0.
3. If the array is empty, omitting the initial value throws a fatal `TypeError: Reduce of empty array with no in`

```
const cart = [
  { item: "Course", price: 49 },
  { item: "Book", price: 29 },
  { item: "Stickers", price: 10 }
];

const totalCartPrice = cart.reduce((acc, currentItem) => {
  return acc + currentItem.price;
}, 0); // 0 is the initial accumulator value

console.log(`Cart total: $${totalCartPrice}`); // "$88"

// Grouping transaction types:
const transactions = [
  { type: "income", amount: 2500 },
  { type: "expense", amount: 450 },
```

```
  { type: "income", amount: 1200 }
];

const balanceSummary = transactions.reduce((acc, t) => {
  acc[t.type] = (acc[t.type] || 0) + t.amount;
  return acc;
}, { income: 0, expense: 0 }); // Initial accumulator is a blank dictionary template
```

Code Listing 8.3: Summing and grouping with reduce

8.4 Searching: `find`, `findIndex`, `some`, and `every`

Table 8.1: Array Searching Methods

Method	Returns	Stops Iterating When	Empty Array Behavior
`find()`	First matched element / `undefined`	Callback returns `true`	Returns `undefined`
`findIndex()`	First matched index / `-1`	Callback returns `true`	Returns `-1`
`some()`	`true` or `false`	Callback returns `true`	Returns `false`
`every()`	`true` or `false`	Callback returns `false`	Returns `true`

```
const members = [
  { id: "M1", name: "Sam", role: "user" },
  { id: "M2", name: "Fedi", role: "admin" },
  { id: "M3", name: "Taylor", role: "user" }
];

const admin = members.find(m => m.role === "admin");
console.log(admin.name); // "Fedi"

const hasAdmin = members.some(m => m.role === "admin"); // true
const allAreAdmins = members.every(m => m.role === "admin"); // false
```

Code Listing 8.4: Array search utilities

8.5 Sorting Arrays: Mutating vs Non-Mutating APIs

The legacy `sort()` method modifies the original array in place. To preserve the original array, modern JavaScript (ES2023+) provides `toSorted()`, which returns a sorted clone:

```
const numbers = [10, 5, 40, 25, 100, 1];

// DANGER: Lexicographical sort by default
numbers.sort();
console.log(numbers); // [1, 10, 100, 25, 40, 5] (Sorted as strings!)

// CORRECT: Numeric sorting using custom comparison callback
const sortedCloned = numbers.toSorted((a, b) => a - b); // ES2023 Non-mutating
```

```
console.log(sortedCloned); // [1, 5, 10, 25, 40, 100]
console.log(numbers);      // [1, 10, 100, 25, 40, 5] (Original preserved!)
```

Code Listing 8.5: Sorting with Comparison Callbacks

8.6 Chaining Array Pipelines

Method chaining allows you to feed the output of one array transformation directly into the input of the next, creating a declarative read-only pipeline:

```
const rawUsers = [
  { name: "fedi", role: "engineer", salary: 95000, active: true },
  { name: "alex", role: "intern", salary: 35000, active: true },
  { name: "sam", role: "designer", salary: 80000, active: false },
  { name: "jordan", role: "engineer", salary: 110000, active: true }
];

// Goal: Total salary payout for ACTIVE ENGINEERS
const totalActiveEngSalary = rawUsers
  .filter(user => user.active && user.role === "engineer")
  .map(user => user.salary)
  .reduce((total, sal) => total + sal, 0);

console.log(`Engineering Payroll: $${totalActiveEngSalary.toLocaleString()}`);
// "Engineering Payroll: $205,000"
```

Code Listing 8.6: Declarative Data Pipeline

MINI CHALLENGE

The Word Frequency Counter. Write a function `wordFrequency(text)` that takes a string, cleans it to lowercase, splits into words, and uses `reduce` to return an object mapping each word to the number of times it appears in the text. Example: `"Fedi loves JavaScript and Fedi loves code"` produces `{ fedi: 2, loves: 2, javascript: 1, and: 1, code: 1}`.

8.7 Exercises

E1. [Level 1] Given `[1, 2, 3, 4, 5]`, use `map` to produce an array of their cubes (x^3).

E2. [Level 1] Given an array of names, use `filter` to keep only names with 5 or more characters.

E3. [Level 2] Given an array of student objects `{ name, score }`, compute the average class score using `reduce`.

E4. [Level 2] Sort an array of products by price in ascending order without mutating the original array.

E5. [Level 3] Implement your own custom version of `myMap(arr, callback)` from scratch using a `for` loop.

E6. [Level 4] Implement a custom `myReduce(arr, callback, initialValue)` function that replicates native `Array.prototype.reduce` behavior.

8.8 Quiz

Q1. What is the definition of a Higher-Order Function?
Q2. What happens if you call `[10, 5, 20].sort()` without a comparison callback?
Q3. How does `map()` differ from `forEach()`?
Q4. What does `some()` return when tested on an empty array? What about `every()`?
Q5. Why is providing an initial value to `reduce()` recommended best practice?

CHAPTER SUMMARY

- HOFs take functions as arguments or return functions; callbacks enable declarative pipelines.
- `map` transforms items $1 \rightarrow 1$; `filter` selects matching subsets.
- `reduce` folds collections into single outputs (sums, tallies, grouped dictionaries).
- `sort` sorts lexicographically by default; always pass `(a, b) => a - b` for numeric sorting.
- Method chaining transforms raw data into final answers in clean, readable pipelines.

INTERVIEW CORNER

Q: What is the difference between `map` and `forEach`? — `forEach` executes a callback for each item solely for side effects and returns `undefined`. `map` returns a brand new array containing transformed elements without mutating the original.

Q: How would you flatten a multi-dimensional array using `reduce`? — `arr.reduce((acc, val) => acc.concat(Array.isArray(val) ? flatten(val) : val), [])` or modern `arr.flat(Infinity)`.

Q: What are the ES2023 non-mutating equivalents for reverse() and splice()? — Use `toReversed()` instead of `reverse()`, and `toSpliced()` instead of `splice()`. They return copy instances rather than modifying the target in place.

9 MODULE

DOM Manipulation and Events

Until now, our JavaScript has lived in the browser console. In this module, JavaScript comes alive in the browser window. The **Document Object Model (DOM)** is the programming interface that connects your JavaScript code to the live HTML and CSS of a webpage. By manipulating the DOM, responding to user events, and leveraging modern browser observers, you turn static documents into dynamic web applications. Mastering the DOM tree, event propagation phases, query selection performance, and observers is essential for building high-performance, responsive web interfaces.

LEARNING OBJECTIVES

- Understand the hierarchical tree structure of the DOM and how browsers parse markup
- Query elements using `querySelector`, `querySelectorAll`, and legacy methods, and understand the difference between Live HTMLCollections and Static NodeLists
- Modify text, HTML structures, attributes, and CSS classes dynamically
- Create, append, and remove elements efficiently using Document Fragments
- Master event propagation (capturing, targeting, and bubbling phases) and distinguish between `e.target` and `e.currentTarget`
- Implement high-performance Event Delegation and understand event cancellation
- Build event optimization strategies: debouncing and throttling
- Dispatch custom application events with the `CustomEvent` API
- Leverage modern browser observers: `IntersectionObserver` and `MutationObserver`

9.1 What Is the DOM?

When a browser loads an HTML document, it parses the markup into a live tree of objects called the **DOM Tree**. Every HTML tag becomes a DOM node object that JavaScript can inspect, modify, move, or destroy in real time.

9.2 Selecting Elements: Live Collections vs Static Lists

There are two primary groups of selector APIs:

1. **Legacy Selectors**: `getElementById`, `getElementsByClassName`, `getElementsByTagName`. They return a live `HTMLCollection` that automatically syncs with DOM mutations.
2. **Modern Selectors**: `querySelector` (first match) and `querySelectorAll` (all matches). They return a static `NodeList` that acts as a snapshot and does not update when elements are added or removed.

```
// Single element selection (returns first match or null)
```

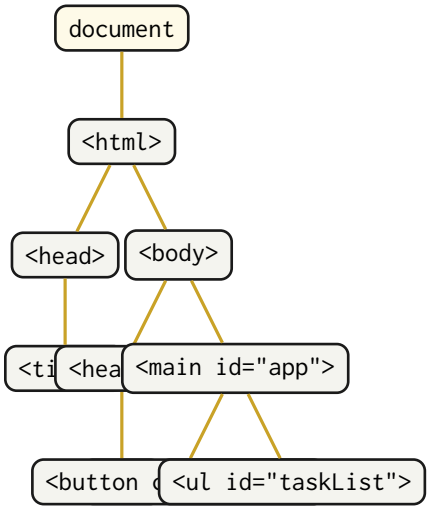


Figure 9.1: The Hierarchical Structure of the DOM Tree

```
const submitButton = document.querySelector("#submit-btn");

// Multiple element selection (returns static NodeList)
const allCards = document.querySelectorAll(".card-item");

// Live HTMLCollection vs Static NodeList:
const parent = document.getElementById("list-parent");
const liveList = parent.getElementsByClassName("item"); // Live HTMLCollection
const staticList = parent.querySelectorAll(".item");     // Static NodeList

console.log(liveList.length);   // e.g., 3
console.log(staticList.length); // e.g., 3

// Append a new item
const newItem = document.createElement("div");
newItem.className = "item";
parent.appendChild(newItem);

console.log(liveList.length);   // 4 (Automatically updated!)
console.log(staticList.length); // 3 (Still 3, snapshot remains unchanged!)
```

Code Listing 9.1: Querying the DOM and Collection Differences

9.3 Changing Content, Attributes, and Classes

```
const heading = document.querySelector("h1");
// textContent: Safest way to update text (automatically escapes HTML)
heading.textContent = "Welcome to FediApp!";

// Managing CSS classes via classList:
const modal = document.querySelector("#modal-dialog");
modal.classList.add("is-visible");
modal.classList.remove("hidden");
```

```
modal.classList.toggle("dark-mode");
```

Code Listing 9.2: Modifying DOM Content and Styles

9.4 Creating and Appending with Document Fragments

Inserting DOM nodes individually inside a loop triggers repeated browser style recalculations and layout passes (reflows). To optimize performance, use a `DocumentFragment` to batch insertions:

```
const container = document.querySelector("#user-list");
const fragment = document.createDocumentFragment(); // In-memory container

const users = ["Fedi", "Sam", "Alex", "Jordan", "Taylor"];

users.forEach(name => {
  const li = document.createElement("li");
  li.textContent = name;
  li.className = "user-item";
  fragment.append(li); // Appends to in-memory fragment (zero browser reflows!)
});

// Single DOM mutation: inserts all 5 items in one render pass!
container.append(fragment);
```

Code Listing 9.3: High-performance batch insertion with DocumentFragment

9.5 Event Propagation: Capture, Target, and Bubble

When an event is triggered on a DOM element, it travels through three distinct phases:

1. **Capturing Phase**: The event trickles down from the root `window` and `document` node to the target element.
2. **Target Phase**: The event fires directly on the element that was clicked or triggered (available via `e.target`).
3. **Bubbling Phase**: The event bubbles back up from the target element to the document root. By default, event listeners listen to this phase.

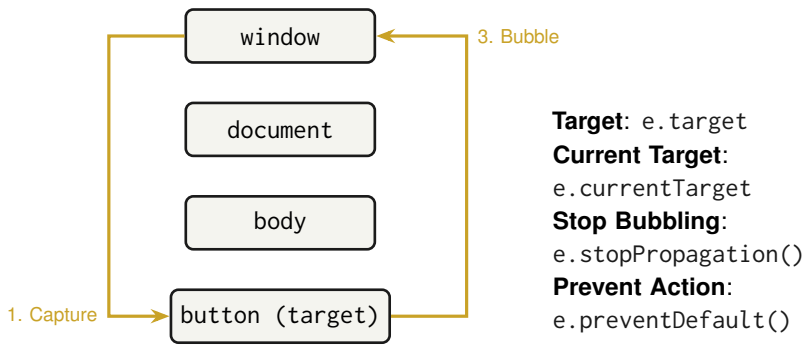


Figure 9.2: DOM Event Propagation Flow

```
const grandparent = document.querySelector("#grandparent");
const parent = document.querySelector("#parent");
const button = document.querySelector("#button");

// Event handler showing target vs currentTarget
parent.addEventListener("click", (e) => {
  console.log(`Clicked element (Target):`, e.target);
  console.log(`Element holding listener (Current Target):`, e.currentTarget);
});

// Adding a capturing listener by setting the third argument to true
grandparent.addEventListener("click", (e) => {
  console.log("Captured at Grandparent!");
}, true); // true = run on capture phase, false/omitted = run on bubbling phase
```

Code Listing 9.4: Event Bubbling vs Capturing

9.6 Event Delegation and Cancellation

Instead of adding listeners to hundreds of child elements (which consumes memory and fails for dynamically added elements), we attach a single listener to a parent element and inspect `e.target` as events bubble up:

```
const taskList = document.querySelector("#taskList");

// A single parent listener handles clicks on all current and FUTURE buttons!
taskList.addEventListener("click", (e) => {
  if (e.target.matches("button.delete-btn")) {
    const parentRow = e.target.closest("li");
    parentRow.remove();
    console.log("Task removed via event delegation!");
  }
});
```

Code Listing 9.5: High-Performance Event Delegation

To control event propagation, use the following methods:

- `e.stopPropagation()`: Prevents the event from continuing to bubble up or capture down.
- `e.stopImmediatePropagation()`: Prevents bubbling and blocks other listeners on the *same* element from executing.
- `e.preventDefault()`: Cancels the browser's default action (e.g., stops links from navigating or forms from reloading the page).

9.7 Event Optimization: Debounce and Throttle

Events like window resizing, mouse movements, or scrolling fire hundreds of times per second. To prevent performance bottlenecks, we wrap event handlers in optimization functions:

```
// DEBOUNCE: Delays execution until 'delay' ms of silence have passed
```

```
function debounce(fn, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => fn.apply(this, args), delay);
  };
}

// THROTTLE: Limits execution to at most once per 'limit' ms
function throttle(fn, limit) {
  let inThrottle = false;
  return function(...args) {
    if (!inThrottle) {
      fn.apply(this, args);
      inThrottle = true;
      setTimeout(() => inThrottle = false, limit);
    }
  };
}

// Usage in scroll events:
window.addEventListener("scroll", throttle(() => {
  console.log("Scroll position tracked at regular intervals");
}, 200));
```

Code Listing 9.6: Implementing Debouncing and Throttling

9.8 Custom Events

Decouple components by emitting and listening to custom domain events:

```
const cartComponent = document.querySelector("#shopping-cart");

// 1. Listen for custom event
document.addEventListener("fedi:cart-updated", (event) => {
  console.log(`Cart total changed: $${event.detail.total}`);
  console.log(`Item count: ${event.detail.itemCount}`);
});

// 2. Dispatch custom event with payload in 'detail' property
function notifyCartUpdate(total, itemCount) {
  const customEv = new CustomEvent("fedi:cart-updated", {
    bubbles: true, // Allow event to bubble up the DOM
    detail: { total, itemCount }
  });
  cartComponent.dispatchEvent(customEv);
}

notifyCartUpdate(149.99, 3);
```

Code Listing 9.7: Creating and Dispatching Custom Events

9.9 Modern Observers: Intersection and Mutation

Modern browsers provide asynchronous observers that run off the main thread to monitor elements without causing performance penalties:

```
const imageObserver = new IntersectionObserver((entries, observer) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      const img = entry.target;
      img.src = img.dataset.src; // Swap data-src to real src to trigger load
      img.classList.add("loaded");
      observer.unobserve(img); // Stop observing once loaded
    }
  });
}, { threshold: 0.1 }); // Fires when 10% of image enters viewport

document.querySelectorAll("img[data-src]").forEach(img => {
  imageObserver.observe(img);
});
```

Code Listing 9.8: Viewport Tracking with IntersectionObserver

```
const parentNode = document.querySelector("#app");

const observer = new MutationObserver((mutationsList) => {
  for (const mutation of mutationsList) {
    if (mutation.type === "childList") {
      console.log("A child node was added or removed.");
    } else if (mutation.type === "attributes") {
      console.log(`The ${mutation.attributeName} attribute was modified.`);
    }
  }
});

// Start observing the target node for configuration changes
observer.observe(parentNode, {
  attributes: true,
  childList: true,
  subtree: true // Monitor all nested descendants as well
});
```

Code Listing 9.9: Subtree Monitoring with MutationObserver

MINI CHALLENGE

The Infinite Scroll Observer. Build an infinite list container with a `<div id="sentinel"></div>` at the bottom. Use an `IntersectionObserver` to detect when the sentinel becomes visible, appending 10 new list items automatically.

9.10 Exercises

E1. [Level 1] Toggle the class `dark-mode` on `document.body` on button click.
E2. [Level 1] Batch append 50 table rows using a `DocumentFragment` with custom cell values.
E3. [Level 2] Implement an event-delegated accordion menu where clicking headers expands content panels.
E4. [Level 2] Dispatch and handle a custom `"user:login"` event carrying user profile metadata.
E5. [Level 3] Implement image lazy loading with a blurry thumbnail placeholder swap using `IntersectionObserver`.
E6. [Level 4] Build a `MutationObserver` that monitors DOM subtree changes to sanitize and strip unauthorized script tags.

9.11 Quiz

Q1. Why is `DocumentFragment` beneficial for batch DOM insertions?
Q2. What are the three phases of DOM event propagation?
Q3. How does Event Delegation minimize memory consumption?
Q4. What property on a `CustomEvent` carries custom data payloads?
Q5. Why is `IntersectionObserver` vastly superior to window `scroll` event listeners?

CHAPTER SUMMARY

- The DOM represents HTML documents as trees of scriptable objects.
- Batch multiple insertions with `DocumentFragment` to minimize reflows.
- Event delegation captures child events as they bubble up to common ancestors.
- Use `CustomEvent` for component decoupling and `IntersectionObserver` for viewport tracking.

INTERVIEW CORNER

Q: What causes DOM Reflow and Repaint, and how do you minimize them? — A **Reflow** (layout calculation) occurs when element geometries, sizes, or positions change. A **Repaint** occurs when visual appearances (colors, visibility) change without altering layout. Minimize them by: batching DOM changes with `DocumentFragment`, modifying CSS classes instead of inline styles, and using `requestAnimationFrame`.

Q: What is the difference between e.target and e.currentTarget? — `e.target` is the deep element where the event originally occurred (e.g., the actual text inside a clicked button). `e.currentTarget` is the element to which the event handler is currently attached (the element executing the listener function).

10 MODULE

Web Accessibility

Building an application that works is the first step. Building one that works **for everyone** is what separates a competent developer from a professional one. **Web Accessibility** (commonly abbreviated **a11y**) is the practice of designing websites and applications so that people with disabilities — including visual, auditory, motor, and cognitive impairments — can perceive, understand, navigate, and interact with the web. Accessibility is not optional: it is a legal requirement in many jurisdictions and an ethical responsibility for every software engineer.

LEARNING OBJECTIVES

- Understand the Web Content Accessibility Guidelines (WCAG) 2.1 and conformance levels (A, AA, AAA)
- Use semantic HTML elements that convey meaning to assistive technologies
- Apply ARIA attributes: `role`, `aria-label`, `aria-describedby`, `aria-live`, `aria-expanded`, `aria-hidden`
- Implement keyboard navigation: `tabindex`, focus management, focus trapping in modals
- Ensure proper color contrast ratios per WCAG AA (4.5:1) and AAA (7:1) standards
- Build accessible forms with linked labels, fieldsets, and live error announcements
- Test accessibility using Lighthouse, axe-core, and manual keyboard audits

10.1 Why Accessibility Matters

According to the World Health Organization, over 1 billion people globally live with some form of disability. Screen reader users, keyboard-only navigators, users with low vision, color blindness, motor impairments, or cognitive differences all rely on accessible software to participate in the digital world.

Three critical reasons to prioritize accessibility:

1. **Legal Compliance**: Laws such as the Americans with Disabilities Act (ADA), the European Accessibility Act (EAA), and Section 508 mandate digital accessibility.
2. **Expanded User Base**: Accessible sites serve a broader audience, including elderly users and people with temporary injuries.
3. **Better Engineering**: Accessible code is well-structured, semantic, testable, and performant. Good accessibility practices improve the experience for *all* users.

10.2 Semantic HTML: The Foundation of Accessibility

Semantic HTML provides built-in accessibility features that `<div>` and `<span>` elements lack entirely. Using the correct element gives assistive technologies the information they need without any extra

JavaScript or ARIA attributes:

Table 10.1: Semantic HTML Elements and Their Accessibility Roles

Semantic Element	Implicit ARIA Role	Purpose
`<button>`	`button`	Interactive click/keypress target (Enter/Space)
`<nav>`	`navigation`	Site navigation region
`<main>`	`main`	Primary content area (one per page)
`<header>`	`banner`	Introductory content or navigation group
`<footer>`	`contentinfo`	Footer with copyright, links, legal
`<article>`	`article`	Self-contained, independently distributable content
`<aside>`	`complementary`	Supplementary content (sidebars, callouts)
`<form>`	`form`	User input collection region

```
<!-- BAD: Non-semantic, inaccessible -->
<div class="btn" onclick="submitForm()">Submit</div>

<!-- GOOD: Semantic, accessible by default -->
<button type="submit">Submit Application</button>

<!-- BAD: Generic div for navigation -->
<div class="nav-links">...</div>

<!-- GOOD: Semantic navigation landmark -->
<nav aria-label="Main Navigation">
  <ul>
    <li><a href="/dashboard">Dashboard</a></li>
    <li><a href="/settings">Settings</a></li>
  </ul>
</nav>
```

Code Listing 10.1: Semantic HTML vs Non-Semantic HTML

COMMON MISTAKE

Never use a `<div>` with an `onclick` handler when a `<button>` would work. A `<div>` is not focusable, does not respond to keyboard Enter/Space by default, and is not announced as interactive by screen readers. Always prefer native HTML interactive elements.

10.3 ARIA Attributes: Enhancing Custom Widgets

When native HTML elements cannot express the required semantics (e.g., custom dropdown menus, tab panels, tree views), use **ARIA** (Accessible Rich Internet Applications) attributes to provide equivalent accessibility information:

```
<!-- Custom Dropdown Trigger -->
<button
  id="menuToggle"
  aria-expanded="false"
  aria-controls="dropdownMenu"
  aria-haspopup="true"
```

```
>
  Menu
</button>
<ul id="dropdownMenu" role="menu" hidden>
  <li role="menuitem"><a href="/profile">Profile</a></li>
  <li role="menuitem"><a href="/settings">Settings</a></li>
  <li role="menuitem"><a href="/logout">Log Out</a></li>
</ul>
```

Code Listing 10.2: ARIA Attributes in Practice

```
const toggle = document.querySelector("#menuToggle");
const menu = document.querySelector("#dropdownMenu");

toggle.addEventListener("click", () => {
  const isOpen = toggle.getAttribute("aria-expanded") === "true";
  toggle.setAttribute("aria-expanded", String(!isOpen));
  menu.hidden = isOpen;
});
```

Code Listing 10.3: Toggling ARIA State with JavaScript

NOTE

The first rule of ARIA: **Do not use ARIA if you can use a native HTML element** that already has the semantics you need. ARIA overrides and supplements, but native elements are always more robust and require less maintenance.

10.4 Keyboard Navigation and Focus Management

Many users navigate entirely with a keyboard. All interactive elements must be **reachable via Tab** and **activatable via Enter or Space**:

```
function openModal(modalEl, triggerEl) {
  modalEl.hidden = false;
  modalEl.setAttribute("aria-hidden", "false");

  // Move focus to the first focusable element inside the modal
  const firstFocusable = modalEl.querySelector(
    'button, [href], input, select, textarea, [tabindex]:not([tabindex="-1"])'
  );
  if (firstFocusable) firstFocusable.focus();

  // Trap focus inside modal
  modalEl.addEventListener("keydown", (e) => {
    if (e.key === "Escape") {
      closeModal(modalEl, triggerEl);
    }

    if (e.key === "Tab") {
      const focusableEls = modalEl.querySelectorAll(
        'button, [href], input, select, textarea, [tabindex]:not([tabindex="-1"])'
```

```
      );
      const firstEl = focusableEls[0];
      const lastEl = focusableEls[focusableEls.length - 1];

      if (e.shiftKey && document.activeElement === firstEl) {
        e.preventDefault();
        lastEl.focus();
      } else if (!e.shiftKey && document.activeElement === lastEl) {
        e.preventDefault();
        firstEl.focus();
      }
    }
  });
}

function closeModal(modalEl, triggerEl) {
  modalEl.hidden = true;
  modalEl.setAttribute("aria-hidden", "true");
  triggerEl.focus(); // Return focus to the element that opened the modal!
}
```

Code Listing 10.4: Focus Management in a Modal Dialog

10.5 Live Regions for Dynamic Content

When JavaScript dynamically updates content (e.g., adding a toast notification, updating a cart total, or displaying a form error), screen readers need to be notified of the change using `aria-live` regions:

```
<!-- HTML: Status announcement region -->
<div id="statusAnnouncer" role="status" aria-live="polite" class="sr-only"></div>

<!-- CSS: Visually hidden but accessible to screen readers -->
<style>
.sr-only {
  position: absolute;
  width: 1px;
  height: 1px;
  padding: 0;
  margin: -1px;
  overflow: hidden;
  clip: rect(0, 0, 0, 0);
  white-space: nowrap;
  border: 0;
}
</style>
```

Code Listing 10.5: Announcing Dynamic Updates to Screen Readers

```
function announceToScreenReader(message) {
  const announcer = document.querySelector("#statusAnnouncer");
  announcer.textContent = ""; // Clear first to re-trigger announcement
```

```
  requestAnimationFrame(() => {
    announcer.textContent = message;
  });
}

// Usage in form validation:
function validateEmail(input) {
  if (!input.validity.valid) {
    const errorMsg = "Please enter a valid email address.";
    input.setAttribute("aria-invalid", "true");
    announceToScreenReader(errorMsg);
  }
}
```

Code Listing 10.6: Announcing Form Validation Errors

10.6 Color Contrast and Visual Accessibility

WCAG defines minimum contrast ratios between foreground text and background colors:

Table 10.2: WCAG Color Contrast Requirements

Level	Normal Text	Large Text	UI Components
AA (minimum)	4.5 : 1	3 : 1	3 : 1
AAA (enhanced)	7 : 1	4.5 : 1	Not defined

BEST PRACTICE

Never convey information through color alone. Always supplement color with text labels, icons, patterns, or underlines. For example, form errors should display a text message **and** a red border, not just a red border.

10.7 Accessible Forms

```
<form id="registrationForm" aria-labelledby="formTitle">
  <h2 id="formTitle">Create Your FediApp Account</h2>

  <div class="form-field">
    <label for="emailInput">Email Address *</label>
    <input
      type="email"
      id="emailInput"
      name="email"
      required
      aria-required="true"
      aria-describedby="emailHint"
      autocomplete="email"
    />
    <span id="emailHint" class="hint">We will never share your email.</span>
```

```
  </div>

  <fieldset>
    <legend>Notification Preferences</legend>
    <label><input type="checkbox" name="notif" value="email" /> Email alerts</label>
    <label><input type="checkbox" name="notif" value="sms" /> SMS alerts</label>
  </fieldset>

  <button type="submit">Create Account</button>
</form>
```

Code Listing 10.7: Building Accessible Forms

10.8 Testing Accessibility

1. **Keyboard-Only Navigation**: Unplug your mouse and navigate entirely with Tab, Shift+Tab, Enter, Space, and Arrow keys. Verify that every interactive element is reachable and that focus order is logical.
2. **Lighthouse Audit**: Open Chrome DevTools, run Lighthouse with the "Accessibility" category. Aim for a score of 90+ on every page.
3. **axe-core Browser Extension**: Install the axe DevTools extension to detect WCAG violations with detailed remediation guidance.
4. **Screen Reader Testing**: Test with a screen reader (VoiceOver on macOS, NVDA on Windows) to verify that announcements, landmarks, and focus management behave correctly.

MINI CHALLENGE

The Accessible Modal Dialog. Build a fully accessible modal that: (1) traps keyboard focus inside the modal when open, (2) closes on Escape key press, (3) returns focus to the trigger button when closed, (4) uses `aria-modal="true"` and `role="dialog"`, and (5) announces the modal title to screen readers using `aria-labelledby`.

10.9 Exercises

E1. [Level 1] Replace three `<div>` elements in an existing page with semantic equivalents (`<nav>`, `<main>`, `<footer>`).

E2. [Level 1] Add `aria-label` to an icon-only button that currently has no visible text.

E3. [Level 2] Implement a skip-to-main-content link that appears on keyboard focus and jumps to `<main>`.

E4. [Level 2] Build an accordion component where each panel header uses `aria-expanded` and `aria-controls`.

E5. [Level 3] Implement a focus-trapping modal dialog with full keyboard support and screen reader announcements.

E6. [Level 4] Build an accessible autocomplete search input with `role="combobox"`, `aria-activedescendant`, and keyboard arrow navigation through suggestions.

10.10 Quiz

Q1. What is the first rule of ARIA?
Q2. What minimum color contrast ratio does WCAG AA require for normal text?
Q3. Why should focus be returned to the trigger element when a modal dialog closes?
Q4. What is the difference between `aria-live="polite"` and `aria-live="assertive"`?
Q5. Why is the `.sr-only` CSS pattern used instead of `display: none` for screen reader content?

CHAPTER SUMMARY

- Semantic HTML provides built-in accessibility; prefer native elements over ARIA-enhanced `<div>` widgets.
- Use ARIA attributes (`aria-expanded`, `aria-live`, `aria-label`) only when native semantics are insufficient.
- Trap keyboard focus inside modal dialogs and return focus to the trigger on close.
- Test with keyboard-only navigation, Lighthouse audits, axe-core, and real screen readers.

INTERVIEW CORNER

Q: How do you make a custom dropdown menu accessible? — Use `role="menu"` on the container, `role="menuitem"` on each option, `aria-expanded` on the trigger button, `aria-haspopup="true"`, and manage keyboard navigation with Arrow keys. Trap focus inside the menu when open and close on Escape.

Q: What is the Accessibility Tree? — The Accessibility Tree is a parallel tree structure derived from the DOM Tree that browsers expose to assistive technologies. It contains only semantically meaningful nodes with their roles, names, states, and properties, stripping out purely visual elements.

11 MODULE

Debugging and Error Handling

Programming is largely the art of finding and fixing mistakes. Novice programmers fear errors and view them as personal failures; senior engineers welcome errors as precise diagnostic reports provided by the runtime engine. In this module, you will master the tools and mental models used by senior engineers to inspect, diagnose, profile memory, and gracefully handle exceptions in complex applications. You will learn the mechanics of the Call Stack, async error handling, global crash prevention, and memory profiling.

LEARNING OBJECTIVES

- Decode runtime error types: `TypeError`, `ReferenceError`, `SyntaxError`, `RangeError`, `URIError`
- Understand Exception Propagation and how errors bubble up the Call Stack
- Handle exceptions gracefully with `try...catch...finally` and master the 'finally' block return overrides
- Master asynchronous error handling for Promises, async/await, and async callbacks
- Create custom application-specific Error class hierarchies with stack trace capturing
- Leverage advanced `console` diagnostics (`table`, `group`, `time`, `trace`)
- Set conditional breakpoints, inspect Call Stacks, and use Source Maps in Chrome DevTools
- Profile memory leaks, heap snapshots, and identify detached DOM nodes

11.1 Reading Error Messages and Stack Traces

Every JavaScript runtime error contains three vital components:

1. **Error Name**: The category of error that occurred.
2. **Error Message**: A description of what went wrong.
3. **Stack Trace**: The exact sequence of active function execution frames that led to the crash, ordered from the point of failure back to the entry script.

Table 11.1: Common JavaScript Error Types

Error Type	Cause
`TypeError`	Attempting an illegal operation on a value (e.g. `null.f()`)
`ReferenceError`	Accessing a variable that has not been declared or is in the TDZ
`SyntaxError`	Invalid JavaScript syntax that fails the compilation parser
`RangeError`	Number or array length outside allowed mathematical boundaries
`URIError`	Passing malformed characters to URI functions like `decodeURI()`

```
1 function fetchUserData(userId) {
```

```
  return parseServerPayload(null);
}

function parseServerPayload(data) {
  return data.profile.name; // Uncaught TypeError: Cannot read properties of null
}

// Call site: fetchUserData(101);
```

Code Listing 11.1: Reading a stack trace

11.2 Exception Propagation and the Call Stack

When an error is thrown, the JavaScript engine halts current function execution and searches the current stack frame for a `try...catch` block. If none is found, the current frame is popped off the Call Stack, and the engine searches the parent frame. This process, known as **Exception Propagation**, continues until the error is caught or the stack is empty, causing a global uncaught exception.

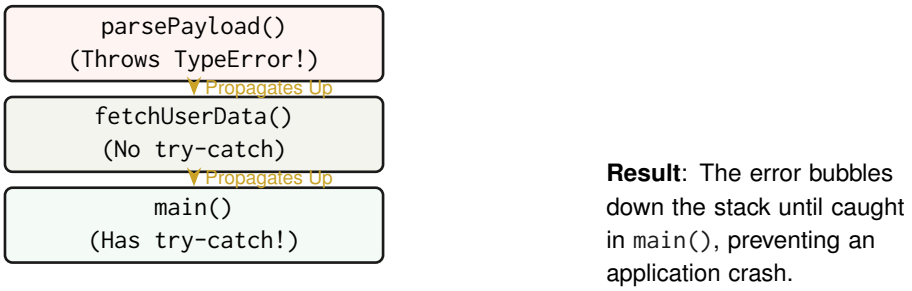


Figure 11.1: Exception Bubbling Up the Call Stack

11.3 The try...catch...finally Statement

The `try...catch...finally` statement handles exceptions. The `finally` block is guaranteed to run **regardless** of whether an error was thrown or caught.

```
function readConfigFile(path) {
  let fileHandle = null;
  try {
    fileHandle = openFile(path);
    return JSON.parse(fileHandle.read());
  } catch (error) {
    console.error(`Failed to read config: ${error.message}`);
    return {};
  } finally {
    if (fileHandle) {
      fileHandle.close(); // Closed regardless of success or failure!
      console.log("File handle cleaned up.");
    }
  }
```

```
}
```

Code Listing 11.2: Standard Exception Handling

COMMON MISTAKE

A `return` statement inside a `finally` block will **override** any `return` or `throw` statements inside the preceding `try` and `catch` blocks. This is a common source of silent failure bugs.

```
function returnTrap() {
  try {
    throw new Error("Initial failure!");
  } catch (err) {
    return "Caught value";
  } finally {
    return "Finally override!"; // Clears the thrown error and the caught return!
  }
}
console.log(returnTrap()); // "Finally override!"
```

Code Listing 11.3: The Finally Return Override Trap

11.4 Asynchronous Error Handling

Synchronous `try...catch` blocks cannot catch errors thrown inside asynchronous contexts (like `setTimeout` or database drivers) because the surrounding `try` block has already finished executing before the asynchronous callback runs.

```
// FAILURE: The error is uncaught!
try {
  setTimeout(() => {
    throw new Error("Delayed crash!"); // Throws after try block has exited
  }, 100);
} catch (err) {
  console.log("Caught:", err); // Never executes!
}

// CORRECT: Wrap the asynchronous operation in a Promise
function delay(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error("Delayed crash!")), ms);
  });
}

delay(100)
  .then(() => console.log("Success"))
  .catch(err => console.log("Successfully caught:", err.message));
```

Code Listing 11.4: The Async Error Catching Trap

To catch unhandled Promise rejections globally in Node.js or browsers:

```
// Browser
window.addEventListener("unhandledrejection", (e) => {
  console.error("Unhandled Promise rejection:", e.reason);
  e.preventDefault(); // Stop default console logging
});

// Node.js
process.on("unhandledRejection", (reason, promise) => {
  console.error("Unhandled rejection at:", promise, "reason:", reason);
  // Recommend crash-and-restart in production
  process.exit(1);
});
```

Code Listing 11.5: Global Uncaught Rejection Listeners

11.5 Creating Custom Error Hierarchies

Creating custom error classes allows you to classify errors and handle them differently based on type.

```
class AppError extends Error {
  constructor(message, statusCode = 500) {
    super(message);
    this.name = this.constructor.name;
    this.statusCode = statusCode;
    // Capture stack trace excluding constructor itself
    Error.captureStackTrace(this, this.constructor);
  }
}

class ValidationError extends AppError {
  constructor(field, message) {
    super(`[Field: ${field}] ${message}`, 400);
    this.field = field;
  }
}

class AuthenticationError extends AppError {
  constructor(message = "Unauthorized access") {
    super(message, 401);
  }
}

// Handling by class type:
try {
  throw new ValidationError("email", "Invalid email address format.");
} catch (err) {
  if (err instanceof ValidationError) {
    console.log(`Validation Failed on ${err.field}: ${err.message}`);
  } else if (err instanceof AuthenticationError) {
    redirectToLogin();
  } else {
    showGenericCrashPage(err);
```

```
  }
}
```

Code Listing 11.6: Custom Application Error Hierarchy

11.6 Advanced Console Diagnostics

```
const users = [
  { id: 1, name: "Fedi", role: "admin", active: true },
  { id: 2, name: "Sam", role: "user", active: false }
];

// 1. Tabular visualization
console.table(users);

// 2. Grouped logging
console.group("FediApp User Sync");
console.log("Connecting to auth server...");
console.log("Token validated.");
console.groupEnd();

// 3. Performance benchmarking
console.time("Array Transformation");
const squares = Array.from({ length: 100000 }, (_, i) => i * i);
console.timeEnd("Array Transformation"); // e.g. "Array Transformation: 4.21ms"

// 4. Call stack trace
function a() { b(); }
function b() { console.trace("How did we get here?"); }
a();
```

Code Listing 11.7: Professional Console Methods

11.7 Browser DevTools, Breakpoints, and Source Maps

DevTools provides a visual debugger to step through execution. Use the `debugger;` statement to pause execution at a specific line programmatically:

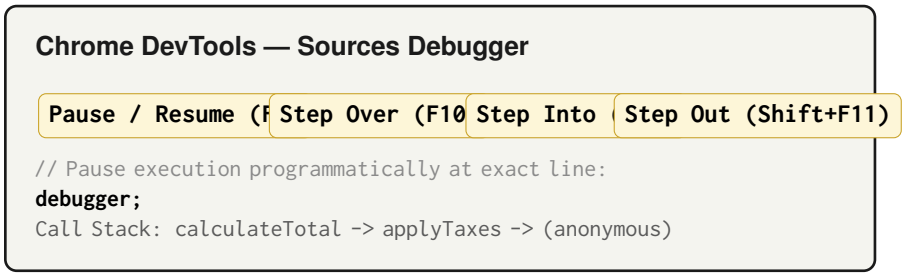

Chrome DevTools — Sources Debugger
Pause / Resume (F
Step Over (F10
Step Into
Step Out (Shift+F11)
// Pause execution programmatically at exact line:
debugger;
Call Stack: calculateTotal -> applyTaxes -> (anonymous)

Figure 11.2: The DevTools Execution Stepping Interface

Step Over (F10) Executes the current line and moves to the next line without stepping inside function calls.

Step Into (F11) Steps inside the function calls on the current line.

Step Out (Shift+F11) Runs the rest of the current function and returns to the caller line.

Source Maps Production code is minified and bundled, making stack traces unreadable. A **Source Map** is a JSON file mapping compiled code back to the original source, allowing you to debug original source code directly in production DevTools.

11.8 Profiling Memory Leaks and Detached DOM Nodes

A **Memory Leak** occurs when memory allocated by an application is no longer needed but is not reclaimed by the garbage collector because references to it remain active.

Common Memory Leak sources:

1. **Forgotten Timers**: `setInterval` callbacks holding references to outer variables.
2. **Uncleared Event Listeners**: Listeners on global objects (`window`, `body`) holding references to destroyed components.
3. **Detached DOM Nodes**: Elements removed from the document tree using `remove()` while a JavaScript variable or array still references them.

```
// LEAK: Removed from DOM, but stored in global array!
const detachedStorage = [];

function createLeakyButton() {
  const btn = document.createElement("button");
  btn.textContent = "Click Me";
  document.body.append(btn);

  // Later removed from visual tree:
  btn.remove();

  // ACCIDENTAL LEAK: Still kept in JavaScript memory!
  detachedStorage.push(btn);
}

// FIX: Clean up references and remove listeners
function cleanupComponent(element, listener) {
  element.removeEventListener("click", listener);
  element.remove();
  // Clear reference to allow GC to clean up
  element = null;
}
```

Code Listing 11.8: Preventing Detached DOM Memory Leaks

MINI CHALLENGE

The Resilient JSON Fetcher. Write a function `safeJsonParse(str, fallback)` that wraps `JSON.parse` in a `try...catch`. If parsing succeeds, return the object. If it fails with a `SyntaxError`, return `fallback` and log a custom warning.

11.9 Exercises

E1. [Level 1] Intentionally trigger and observe a `TypeError` and a `ReferenceError` in DevTools.
E2. [Level 1] Benchmark two loop implementations using `console.time()`.
E3. [Level 2] Set a conditional breakpoint that only pauses when a loop counter reaches `i === 50`.
E4. [Level 2] Take a Heap Snapshot in Chrome DevTools and identify detached DOM nodes.
E5. [Level 3] Write an error boundary wrapper `trySafe(fn)` returning `[result, null]` or `[null, err]`.
E6. [Level 4] Build a global error reporter intercepting `window.onerror` and `unhandledrejection`.

11.10 Quiz

Q1. What are the three primary components of an Error object in JavaScript?
Q2. When does the `finally` block execute in a `try...catch...finally` statement?
Q3. What are detached DOM nodes and why do they cause memory leaks?
Q4. How do you take a Heap Snapshot in Chrome DevTools?
Q5. What keyword pauses execution directly in the browser debugger?

CHAPTER SUMMARY

- Stack traces guide you directly to root causes; read them top to bottom.
- Use DevTools breakpoints to step through active execution contexts.
- Clean up event listeners and timers to prevent memory leaks and detached DOM nodes.
- Extend `Error` to create domain-specific typed exceptions for clean error branching.

INTERVIEW CORNER

Q: What is a detached DOM node memory leak and how do you diagnose it? — A detached DOM node is an element removed from the live document tree that cannot be garbage-collected because a JavaScript variable or closure retains a reference to it. Diagnose it in Chrome DevTools by taking a Heap Snapshot and filtering for "Detached HTMLElement".

Q: How does async/await exception handling differ from Promise catch chains? — Under the hood, both compile to the same microtask schedule. However, async/await allows using standard synchronous `try...catch` blocks around asynchronous actions, yielding much more readable exception handling pipelines compared to nested or chained `.catch()` callbacks.

12 MODULE

Regular Expressions

Text is the universal currency of software. Usernames, email addresses, phone numbers, postal codes, and URLs all arrive as raw text strings. **Regular Expressions** (commonly known as **Regex**) provide a powerful, compact domain-specific language for searching, matching, validating, and replacing patterns inside text. In this module, you will learn to harness regular expressions from basic syntax to advanced lookarounds, understand the string API interactions, and avoid performance disasters like catastrophic backtracking.

LEARNING OBJECTIVES

- Create regular expressions with regex literals (`/pattern/flags`) and `new RegExp()`
- Use character classes (`\d`, `\w`, `\s`, custom brackets `[a-z]`)
- Control repetition with quantifiers (`+`, `*`, `?`, `{n,m}`)
- Enforce string boundaries with anchors (`^`, `$`, and word boundary `\b`)
- Master regex flags: global (`g`), case-insensitive (`i`), multiline (`m`), unicode (`u/v`), sticky (`y`)
- Compare string methods: `match()`, `matchAll()`, `search()`, `replace()`, `replaceAll()`, `split()`
- Capture and name sub-patterns using Named Capture Groups (`(?<groupName>...)`) and backreferences
- Master Advanced Assertions: Positive/Negative Lookaheads and Lookbehinds
- Understand and prevent catastrophic backtracking ($O(2^n)$ performance failure)

12.1 Creating Regular Expressions

There are two ways to construct a regular expression in JavaScript:

```
// 1. Regex literal: compiled once when script loads (preferred for fixed patterns)
const jsPattern = /javascript/i; // 'i' flag = case-insensitive

// 2. RegExp constructor: useful when pattern is dynamic (from a variable)
const dynamicKeyword = "fedi";
const dynamicPattern = new RegExp(dynamicKeyword, "i");

console.log(jsPattern.test("I love JavaScript!")); // true
console.log(dynamicPattern.test("Welcome, FediUser!")); // true
```

Code Listing 12.1: Regex literal vs constructor

12.2 Character Classes and Shorthands

Character classes define sets of characters to match at a given position.

Table 12.1: Essential Regular Expression Character Classes

Class	Matches	Opposite
`\d`	Any digit [0-9]	`\D` (any non-digit)
`\w`	Any word character [a-zA-Z0-9_]	`\W` (non-word character)
`\s`	Any whitespace (space, tab, newline)	`\S` (non-whitespace)
`.`	Any single character except newline	—
`[abc]`	Exactly 'a', 'b', or 'c'	`[^abc]` (anything EXCEPT a, b, c)
`[a-z]`	Any lowercase letter from a to z	`[^a-z]`

12.3 Quantifiers and Boundaries

Table 12.2: Regex Quantifiers and Anchors

Token	Meaning
`^` / `$`	Assert start / end of string
`\b`	Word boundary (between a word character and non-word character)
`+`	1 or more times (greedy)
`*`	0 or more times (greedy)
`?`	0 or 1 time (optional)
`+?` / `*?`	Non-greedy / lazy matching
`{n,m}`	Between n and m repetitions

By default, quantifiers are **greedy**: they consume as many characters as possible. Adding a `?` to a quantifier makes it **lazy** (or non-greedy), causing it to stop at the first match.

```
const html = '<div class="header">FediApp</div>';

// Greedy matches everything up to the LAST '>'
console.log(html.match(/<.+>/)[0]); // '<div class="header">FediApp</div>'

// Lazy matches up to the FIRST '>'
console.log(html.match(/<.+?>/)[0]); // '<div class="header">'
```

Code Listing 12.2: Greedy vs Lazy Matching

12.4 Regex Flags in Depth

Flags modify the matching behavior of the regular expression:

Find all matches rather than stopping at the first.

Enable case-insensitive matching.

Make `^` and `$` match the beginning/end of lines (separated by `\n`), not just the whole string.

Enable full Unicode support. Necessary for correctly matching emojis or wide characters.

Matches only starting from the index indicated by the `lastIndex` property.

12.5 String Methods Supporting Regex

JavaScript strings provide several APIs that integrate with regular expressions:

```
const text = "Contact fedi@example.com or sam@example.com for help.";
const emailPattern = /\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b/g;

// 1. String.prototype.match() - returns all matches as string array (if global)
console.log(text.match(emailPattern));
// ["fedi@example.com", "sam@example.com"]

// 2. String.prototype.matchAll() - returns iterator with capture groups
for (const match of text.matchAll(emailPattern)) {
  console.log(`Found email: ${match[0]} at index ${match.index}`);
}

// 3. String.prototype.replace() - replace matched patterns
const hiddenEmails = text.replace(emailPattern, "[REDACTED]");
console.log(hiddenEmails); // "Contact [REDACTED] or [REDACTED] for help."

// 4. String.prototype.split() - split string by regex pattern
const csv = "Apple, Banana; Orange | Grapes";
const fruits = csv.split(/\s*[,;|]\s*/);
console.log(fruits); // ["Apple", "Banana", "Orange", "Grapes"]
```

Code Listing 12.3: Regex and String Methods

12.6 Named Capture Groups and Backreferences

Capture groups allow you to isolate and extract sub-patterns from a match. Using `(?<name>...)` assigns a key to the capture group, making extraction readable.

```
const dateString = "2026-08-16";
const datePattern = /^(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})$/;

const match = dateString.match(datePattern);
if (match) {
  const { year, month, day } = match.groups;
  console.log(`Year: ${year}, Month: ${month}, Day: ${day}`);
}
```

Code Listing 12.4: Named Capture Groups

Backreferences allow you to refer to previous capture groups within the same pattern or in the replacement string:

```
// Match duplicate consecutive words (e.g. "the the")
```

```
const doubleWordPattern = /\b(?<word>\w+)\s+\k<word>\b/i; // \k<word> matches the group
    ↪ content
console.log(doubleWordPattern.test("Is the the project ready?")); // true

// Using backreferences in replacements
const reordered = "Yahiyaoui, Fedi".replace(/(\w+),\s*(\w+)/, "$2 $1");
console.log(reordered); // "Fedi Yahiyaoui"
```

Code Listing 12.5: Using Backreferences

12.7 Lookahead and Lookbehind Assertions

Lookarounds are zero-width assertions. They inspect characters before or after the cursor without adding them to the final match.

Table 12.3: Lookaround Assertions in Modern JavaScript

Syntax	Name	Meaning
`(?=...)`	Positive Lookahead	Assert pattern matches immediately to the right
`(?!...)`	Negative Lookahead	Assert pattern does not match to the right
`(?<=...)`	Positive Lookbehind	Assert pattern matches immediately to the left
`(?<!...)`	Negative Lookbehind	Assert pattern does not match to the left

```
// Password Validator: Must contain at least 1 uppercase and 1 digit, min length 8:
const strongPassword = /^(?=.*[A-Z])(?=.*\d)[A-Za-z\d@$!%*?&]{8,}$/;

console.log(strongPassword.test("Fedi2026Password")); // true
console.log(strongPassword.test("onlylowercase"));     // false
```

Code Listing 12.6: Password Validation with Lookaheads

12.8 Catastrophic Backtracking and Regex Performance

Regular expression engines in JavaScript use a backtracking algorithm. When matching fails, the engine undoes its steps and tries alternative combinations. If a pattern has nested greedy quantifiers (e.g. `/(a+)+$/`), a non-matching string of length n can require $O(2^n)$ steps to fail, freezing the main execution thread completely.

```
// DANGEROUS PATTERN: Nested quantifiers
const badPattern = /^(a+)+$/;

// Matching works instantly
console.log(badPattern.test("aaaaaaaaaaaaaaaaaaaa")); // true (20 'a's)

// Non-matching string of same length takes seconds/minutes to evaluate!
const start = performance.now();
badPattern.test("aaaaaaaaaaaaaaaaaaaaX"); // 'X' prevents match, forcing backtracking
```

```
10 console.log(`Took: ${performance.now() - start}ms`); // Exponentially slow!
```

Code Listing 12.7: Example of Catastrophic Backtracking

BEST PRACTICE

To avoid catastrophic backtracking: (1) Avoid nesting quantifiers like * or +. (2) Make patterns mutually exclusive (do not match overlapping characters). (3) Set strict anchors (^, $) to fail early. (4) Benchmark your regular expressions with mock non-matching strings before using them in production.

FSA representation of regex:
/\d[a-z]/
Matches: 1a, 9z, 3b
State q0: start state
State q2: accept/double-ring state

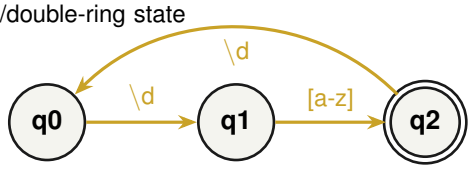


Figure 12.1: Finite State Automaton Diagram for a Simple Pattern

MINI CHALLENGE

The Email Extractor and Parser. Write a script that uses `matchAll()` to find all valid email addresses in a long block of text. Extract the username and domain of each address using named capture groups and return them in an array of objects.

12.9 Exercises

E1. [Level 1] Write a regex that tests if a string contains only letters and spaces.
E2. [Level 1] Use `replace()` with a regex to remove non-alphanumeric characters from `"Hello,␣\#Fedi\_2026!"`.
E3. [Level 2] Write a regex to extract all hashtag words from a string (`"\#javascript␣is␣\#awesome"`).
E4. [Level 2] Write a function `maskCreditCard(str)` masking all but the last 4 digits.
E5. [Level 3] Parse a standard URL into its constituent parts (`protocol`, `hostname`, `pathname`) using named capture groups.
E6. [Level 4] Write a mini Markdown-to-HTML parser using regex replacement to convert bold (`**text**`), italics (`*text*`), and links (`[text](url)`).

12.10 Quiz

Q1. What is the difference between `\d+` and `\d*`?
Q2. What do the anchors `^` and `$` do?
Q3. What flag makes a regex search global across the entire string?
Q4. How do Positive and Negative Lookaheads work in password validation?
Q5. Why is `RegExp.prototype.test()` faster than `String.prototype.match()` for simple boolean checks?

CHAPTER SUMMARY

- Regular expressions match text patterns concisely with literals (`/pattern/`) or constructor (`new RegExp`).
- Character classes (`\d`, `\w`, `\s`) and ranges (`[a-z]`) define allowed character types.
- Use lookarounds (`(?=...)`) to enforce multiple validation conditions simultaneously.
- Always anchor validation patterns with `^` and `$` to prevent partial substring matches.

INTERVIEW CORNER

Q: What is regex catastrophic backtracking and how do you avoid it? — Catastrophic backtracking occurs with nested quantifiers (e.g. `(a+)+$`) where a non-matching string causes the regex engine to explore exponential execution paths ($O(2^n)$), freezing the thread. Avoid nested quantifiers and use atomic grouping or strict anchors.

Q: What is the difference between match() and matchAll()? — `match()` with a global flag returns an array of matched strings but loses capture groups and indices. `matchAll()` returns an iterator yielding detailed match objects for every match, preserving capture groups and positional indices even for global searches.

13 MODULE

Form Validation

Forms are the primary gateway between users and applications. Login forms, registration forms, search queries, payment checkouts, and contact forms all collect user input that must be validated before processing. Invalid or malicious input can cause server crashes, data corruption, and security vulnerabilities. In this module, you will master both the browser's built-in **Constraint Validation API** and custom JavaScript validation patterns to build robust, accessible, real-time form validation systems.

LEARNING OBJECTIVES

- Use HTML5 validation attributes: `required`, `minlength`, `maxlength`, `pattern`, `type`
- Access the Constraint Validation API: `validity`, `checkValidity()`, `setCustomValidity()`
- Implement real-time field validation with `input` and `blur` event listeners
- Display inline error messages with accessible `aria-describedby` associations
- Build multi-field cross-validation logic (password confirmation, conditional fields)
- Sanitize and escape user input to prevent XSS injection attacks
- Submit validated form data asynchronously with `fetch()` and FormData

13.1 HTML5 Built-In Validation Attributes

HTML5 provides declarative validation attributes that browsers enforce automatically when a form is submitted:

Table 13.1: HTML5 Validation Attributes

Attribute	Purpose	Example
`required`	Field must not be empty	`<input required />`
`type="email"`	Must match email format	`<input type="email"/>`
`type="url"`	Must match URL format	`<input type="url"/>`
`minlength` / `maxlength`	Character count bounds	`minlength="8"maxlength="64"`
`min` / `max`	Numeric range bounds	`min="1"max="100"`
`pattern`	Must match a regex pattern	`pattern="[A-Za-z]{3,}"`
`step`	Numeric step increment	`step="0.01"`

```
<form id="registrationForm" novalidate>
  <div class="form-field">
    <label for="usernameField">Username (3-20 alphanumeric characters)</label>
    <input
      type="text"
      id="usernameField"
```

```
      name="username"
      required
      minlength="3"
      maxlength="20"
      pattern="^[a-zA-Z][a-zA-Z0-9_]{2,19}$"
      autocomplete="username"
    />
    <span class="error-msg" id="usernameError"></span>
  </div>

  <div class="form-field">
    <label for="emailField">Email Address</label>
    <input
      type="email"
      id="emailField"
      name="email"
      required
      autocomplete="email"
      aria-describedby="emailError"
    />
    <span class="error-msg" id="emailError"></span>
  </div>

  <div class="form-field">
    <label for="passwordField">Password (min 8 characters)</label>
    <input
      type="password"
      id="passwordField"
      name="password"
      required
      minlength="8"
      autocomplete="new-password"
    />
    <span class="error-msg" id="passwordError"></span>
  </div>

  <button type="submit" class="btn btn-primary">Create Account</button>
</form>
```

Code Listing 13.1: HTML5 Validation Attributes in Action

13.2 The Constraint Validation API

Every form input element has a `validity` property that returns a `ValidityState` object with boolean flags for each type of validation failure:

```
function getFieldError(input) {
  const v = input.validity;

  if (v.valueMissing) return `${input.name} is required.`;
  if (v.typeMismatch) return `Please enter a valid ${input.type}.`;
  if (v.patternMismatch) return `${input.name} format is invalid.`;
  if (v.tooShort) return `Must be at least ${input.minLength} characters.`;
```

Table 13.2: ValidityState Properties

Property	True When
`valueMissing`	Required field is empty
`typeMismatch`	Value does not match the `type` (e.g. invalid email)
`patternMismatch`	Value does not match the `pattern` regex
`tooShort`	Value is shorter than `minlength`
`tooLong`	Value exceeds `maxlength`
`rangeUnderflow`	Numeric value is below `min`
`rangeOverflow`	Numeric value exceeds `max`
`customError`	A custom error was set via `setCustomValidity()`
`valid`	All constraints are satisfied

```
  if (v.tooLong) return `Must be no more than ${input.maxLength} characters.`;
  if (v.rangeUnderflow) return `Value must be at least ${input.min}.`;
  if (v.rangeOverflow) return `Value must be at most ${input.max}.`;

  return ""; // No error
}
```

Code Listing 13.2: Using the Constraint Validation API

13.3 Real-Time Inline Validation

Best practice is to validate fields on `blur` (when the user leaves a field) and on `input` (as they type after the first error), displaying inline error messages:

```
const form = document.querySelector("#registrationForm");

function validateField(input) {
  const errorSpan = document.querySelector(`#${input.id}Error`);
  const errorMsg = getFieldError(input);

  if (errorMsg) {
    input.classList.add("field-invalid");
    input.classList.remove("field-valid");
    input.setAttribute("aria-invalid", "true");
    errorSpan.textContent = errorMsg;
  } else {
    input.classList.remove("field-invalid");
    input.classList.add("field-valid");
    input.removeAttribute("aria-invalid");
    errorSpan.textContent = "";
  }

  return errorMsg === "";
}

// Validate on blur (first interaction) and input (subsequent typing)
form.querySelectorAll("input").forEach(input => {
  input.addEventListener("blur", () => validateField(input));
```

```
  input.addEventListener("input", () => {
    if (input.classList.contains("field-invalid")) {
      validateField(input); // Re-validate as they type to clear error
    }
  });
});
```

Code Listing 13.3: Real-Time Inline Validation Controller

13.4 Cross-Field Validation: Password Confirmation

```
const password = document.querySelector("#passwordField");
const confirm = document.querySelector("#confirmPasswordField");

function validatePasswordMatch() {
  if (confirm.value !== password.value) {
    confirm.setCustomValidity("Passwords do not match.");
  } else {
    confirm.setCustomValidity(""); // Clear custom error
  }
  validateField(confirm);
}

confirm.addEventListener("input", validatePasswordMatch);
password.addEventListener("input", () => {
  if (confirm.value.length > 0) validatePasswordMatch();
});
```

Code Listing 13.4: Password Confirmation Cross-Validation

13.5 Form Submission with Validation Gate

```
form.addEventListener("submit", async (e) => {
  e.preventDefault(); // Prevent default browser submission

  // Validate all fields
  const inputs = form.querySelectorAll("input");
  let allValid = true;
  inputs.forEach(input => {
    if (!validateField(input)) allValid = false;
  });

  if (!allValid) {
    // Focus the first invalid field for accessibility
    const firstInvalid = form.querySelector(".field-invalid");
    if (firstInvalid) firstInvalid.focus();
    return;
  }

```

```
  // All valid: submit asynchronously
  const formData = new FormData(form);

  try {
    const response = await fetch("/api/register", {
      method: "POST",
      body: formData
    });

    if (!response.ok) throw new Error("Registration failed");
    const data = await response.json();
    console.log("Registration successful:", data);
  } catch (err) {
    console.error("Submission error:", err.message);
  }
});
```

Code Listing 13.5: Gated Form Submission with fetch()

13.6 Input Sanitization and XSS Prevention

COMMON MISTAKE

Never inject raw user input into the DOM using `innerHTML`. This creates Cross-Site Scripting (XSS) vulnerabilities where attackers can execute arbitrary JavaScript in your users' browsers. Always use `textContent` for text and sanitize HTML input with libraries like DOMPurify.

```
const userInput = '<img src=x onerror="alert(document.cookie)">';

// DANGEROUS: Executes malicious script!
// element.innerHTML = userInput;

// SAFE: Renders as plain text, no script execution
element.textContent = userInput;
// Displays literal: <img src=x onerror="alert(document.cookie)">
```

Code Listing 13.6: Safe Output: textContent vs innerHTML

MINI CHALLENGE

The Multi-Step Registration Wizard. Build a 3-step registration form where: Step 1 collects name and email, Step 2 collects password with strength meter, Step 3 shows a confirmation summary. Each step must validate before allowing the user to proceed to the next step. Use `aria-current="step"` for screen reader navigation.

13.7 Exercises

E1. [Level 1] Add `required` and `type="email"` validation to an existing form and test the browser's default error popups.

E2. [Level 1] Write a function that checks if a text input's value length is between 3 and 50 characters.
E3. [Level 2] Implement inline error messages that appear on `blur` with accessible `aria-describedby` associations.
E4. [Level 2] Build a password strength meter that evaluates length, uppercase, numbers, and symbols.
E5. [Level 3] Implement a form with conditional validation: show a "Company Name" field only when the user selects "Business" account type, and make it required only when visible.
E6. [Level 4] Build a complete registration form with real-time validation, password confirmation, XSS sanitization, and asynchronous `fetch()` submission.

13.8 Quiz

Q1. What does the `novalidate` attribute on a `<form>` element do?
Q2. What is the difference between `checkValidity()` and `reportValidity()`?
Q3. How does `setCustomValidity("")` differ from `setCustomValidity("error message")`?
Q4. Why should you validate on `blur` for first interaction and `input` for subsequent corrections?
Q5. What is the XSS risk of using `innerHTML` with user-supplied data?

CHAPTER SUMMARY

- HTML5 validation attributes provide declarative, zero-JavaScript form constraints.
- The Constraint Validation API (`validity`, `setCustomValidity`) enables programmatic validation control.
- Validate on `blur` first, then on `input` to provide responsive inline error feedback.
- Always use `textContent` instead of `innerHTML` for user-supplied data to prevent XSS attacks.

INTERVIEW CORNER

Q: Should you validate on the client, the server, or both? — Always validate on **both**. Client-side validation provides instant user feedback and reduces unnecessary server requests. Server-side validation is the security gate — client-side validation can be bypassed by disabling JavaScript or sending raw HTTP requests. Never trust client-side validation as a security measure.

Q: What is the difference between sanitization and validation? — **Validation** checks whether input meets expected criteria (format, length, type). **Sanitization** modifies input to remove or escape dangerous characters (stripping HTML tags, escaping SQL, encoding entities). Both are required for secure applications.

14 MODULE

Dates and Time

Time is one of the most deceptively complex domains in programming. Time zones, daylight saving transitions, calendar variations, leap seconds, and locale-specific formatting rules create a minefield of subtle bugs. JavaScript provides the built-in `Date` object for time manipulation, and the modern `Intl` API for internationalized, locale-aware formatting. In this module, you will learn to create, manipulate, compare, format, and calculate with dates safely and correctly.

LEARNING OBJECTIVES

- Create Date objects using constructors, timestamps, and ISO 8601 strings
- Extract and modify date components (year, month, day, hours, minutes, seconds)
- Navigate common gotchas: zero-indexed months, UTC vs local time, timezone offsets
- Perform date arithmetic: add/subtract days, calculate time differences
- Format dates for any locale using the modern `Intl.DateTimeFormat` API
- Display relative time strings with `Intl.RelativeTimeFormat`
- Build practical utilities: countdown timers, age calculators, and scheduling helpers

14.1 Creating Date Objects

The `Date` object represents a single moment in time as the number of milliseconds elapsed since the **Unix Epoch**: January 1, 1970, 00:00:00 UTC.

```
// 1. Current date and time
const now = new Date();
console.log(now); // e.g. "Sat Aug 16 2026 20:30:00 GMT+0100"

// 2. From ISO 8601 string (always parsed as UTC when 'Z' is included)
const launch = new Date("2026-09-01T10:00:00Z");

// 3. From individual components (WARNING: month is ZERO-INDEXED!)
const birthday = new Date(2000, 2, 15); // March 15, 2000 (month 2 = March!)

// 4. From Unix timestamp in milliseconds
const fromTimestamp = new Date(1724868600000);

// 5. Current timestamp (milliseconds) without creating an object
const timestampMs = Date.now(); // Fastest method for benchmarking
```

Code Listing 14.1: Creating Date Objects

COMMON MISTAKE

JavaScript months are **zero-indexed**: January = 0, February = 1, ..., December = 11. This is one of the most common sources of off-by-one date bugs. Always double-check month values when constructing dates manually.

14.2 Reading Date Components

Table 14.1: Date Getter Methods (Local Time)

Method	Returns	Range
`getFullYear()`	4-digit year	e.g. 2026
`getMonth()`	Month (0-indexed!)	0–11
`getDate()`	Day of month	1–31
`getDay()`	Day of week (0 = Sunday)	0–6
`getHours()`	Hours (24h format)	0–23
`getMinutes()`	Minutes	0–59
`getSeconds()`	Seconds	0–59
`getTime()`	Unix timestamp (ms)	Large integer

```
const eventDate = new Date("2026-12-25T18:00:00");
console.log(eventDate.getFullYear()); // 2026
console.log(eventDate.getMonth());    // 11 (December is index 11, not 12!)
console.log(eventDate.getDate());     // 25
console.log(eventDate.getDay());      // 5 (Friday)
console.log(eventDate.getHours());    // 18
```

Code Listing 14.2: Extracting Date Components

14.3 UTC vs Local Time

Every `Date` getter has a UTC counterpart (`getUTCFullYear()`, `getUTCMonth()`, etc.) that returns the value in Coordinated Universal Time rather than the browser's local timezone:

```
const meeting = new Date("2026-08-16T14:00:00Z"); // 2pm UTC

// Local time (depends on browser timezone)
console.log(meeting.getHours());    // 15 in UTC+1 (Tunis), 10 in UTC-4 (New York)

// UTC time (always consistent)
console.log(meeting.getUTCHours()); // 14 (always 14, regardless of timezone)

// Timezone offset in minutes
const offsetMinutes = meeting.getTimezoneOffset();
console.log(`Offset from UTC: ${-offsetMinutes / 60} hours`); // e.g. "1 hours"
```

Code Listing 14.3: UTC vs Local Time Comparison

14.4 Date Arithmetic and Durations

Since dates are stored internally as millisecond timestamps, arithmetic is performed by adding or subtracting milliseconds:

```
const MS_PER_SECOND = 1000;
const MS_PER_MINUTE = MS_PER_SECOND * 60;
const MS_PER_HOUR   = MS_PER_MINUTE * 60;
const MS_PER_DAY    = MS_PER_HOUR * 24;

// Add N days to a date (returns a NEW Date, does not mutate!)
function addDays(date, days) {
  return new Date(date.getTime() + days * MS_PER_DAY);
}

const today = new Date("2026-08-16");
const nextWeek = addDays(today, 7);
console.log(nextWeek.toISOString()); // "2026-08-23T00:00:00.000Z"

// Calculate difference between two dates in days
function daysBetween(dateA, dateB) {
  const diffMs = Math.abs(dateB.getTime() - dateA.getTime());
  return Math.floor(diffMs / MS_PER_DAY);
}

const projectStart = new Date("2026-01-15");
const projectEnd = new Date("2026-08-16");
console.log(`Project duration: ${daysBetween(projectStart, projectEnd)} days`);
// "Project duration: 213 days"
```

Code Listing 14.4: Date Arithmetic Utilities

BEST PRACTICE

Always use `getTime()` for date comparisons and arithmetic instead of string comparison. Two Date objects with identical times are **never equal** with === because they are different object references. Compare their `.getTime()` values instead.

14.5 The Intl.DateTimeFormat API

The `Intl.DateTimeFormat` API provides locale-aware, culturally correct date and time formatting without any external libraries:

```
const eventDate = new Date("2026-08-16T14:30:00Z");

// English (US) format
const usFormatter = new Intl.DateTimeFormat("en-US", {
  weekday: "long",
  year: "numeric",
  month: "long",
  day: "numeric",
```

```
  hour: "2-digit",
  minute: "2-digit",
  timeZone: "America/New_York"
});
console.log(usFormatter.format(eventDate));
// "Sunday, August 16, 2026, 10:30 AM"

// Arabic (Tunisia) format
const arFormatter = new Intl.DateTimeFormat("ar-TN", {
  year: "numeric",
  month: "long",
  day: "numeric",
  timeZone: "Africa/Tunis"
});
console.log(arFormatter.format(eventDate));
// Displays date in Arabic script with Tunisian locale conventions

// Japanese format
const jpFormatter = new Intl.DateTimeFormat("ja-JP", {
  year: "numeric",
  month: "short",
  day: "numeric"
});
console.log(jpFormatter.format(eventDate));
// "2026/8/16"
```

Code Listing 14.5: Internationalized Date Formatting

14.6 Relative Time with Intl.RelativeTimeFormat

Modern applications display human-friendly relative timestamps like “3 days ago” or “in 2 hours” instead of raw dates:

```
const rtf = new Intl.RelativeTimeFormat("en", { numeric: "auto" });

console.log(rtf.format(-1, "day"));   // "yesterday"
console.log(rtf.format(-3, "day"));   // "3 days ago"
console.log(rtf.format(1, "hour"));   // "in 1 hour"
console.log(rtf.format(0, "day"));    // "today"

// Utility: compute relative time from now
function timeAgo(pastDate) {
  const diffMs = Date.now() - pastDate.getTime();
  const diffDays = Math.floor(diffMs / MS_PER_DAY);
  const diffHours = Math.floor(diffMs / MS_PER_HOUR);
  const diffMinutes = Math.floor(diffMs / MS_PER_MINUTE);

  const rtf = new Intl.RelativeTimeFormat("en", { numeric: "auto" });
  if (diffDays > 0) return rtf.format(-diffDays, "day");
  if (diffHours > 0) return rtf.format(-diffHours, "hour");
  return rtf.format(-diffMinutes, "minute");
```

```
}
```

Code Listing 14.6: Relative Time Formatting

14.7 Practical Pattern: Countdown Timer

```
function startCountdown(targetDate) {
  const timer = setInterval(() => {
    const remainMs = targetDate.getTime() - Date.now();

    if (remainMs <= 0) {
      clearInterval(timer);
      console.log("Countdown complete!");
      return;
    }

    const days = Math.floor(remainMs / MS_PER_DAY);
    const hours = Math.floor((remainMs % MS_PER_DAY) / MS_PER_HOUR);
    const minutes = Math.floor((remainMs % MS_PER_HOUR) / MS_PER_MINUTE);
    const seconds = Math.floor((remainMs % MS_PER_MINUTE) / MS_PER_SECOND);

    console.log(`${days}d ${hours}h ${minutes}m ${seconds}s remaining`);
  }, 1000);

  return timer; // Return timer ID for cleanup
}

// Count down to New Year 2027
startCountdown(new Date("2027-01-01T00:00:00"));
```

Code Listing 14.7: Building a Countdown Timer

MINI CHALLENGE

The Fedi Age Calculator. Write a function `calculateAge(birthDateStr)` that accepts an ISO date string (e.g. `"2000-03-15"`), calculates the user's exact age in years, months, and days, and returns a formatted string like `"26␣years,␣5␣months,␣1␣day"`. Handle edge cases for February 29 birthdays.

14.8 Exercises

E1. [Level 1] Create a Date for your birthday and log the day of the week you were born using `getDay()`.

E2. [Level 1] Write a function `isWeekend(date)` returning `true` if the date falls on Saturday or Sunday.

E3. [Level 2] Write a function `formatDate(date, locale)` using `Intl.DateTimeFormat` to display a full date string.

E4. [Level 2] Implement `daysBetween(dateA, dateB)` and test it with project start/end dates.

E5. [Level 3] Build a meeting scheduler that accepts a UTC timestamp and converts it into 3 participant time zones (`"America/New\_York"`, `"Europe/London"`, `"Africa/Tunis"`).

E6. [Level 4] Build a full countdown timer component that renders days, hours, minutes, and seconds in the DOM and updates every second using `setInterval`.

14.9 Quiz

Q1. Why are JavaScript months zero-indexed and what bugs does this commonly cause?

Q2. What is the difference between `Date.now()` and `new Date().getTime()`?

Q3. Why should you avoid comparing two Date objects with ===?

Q4. What advantage does `Intl.DateTimeFormat` offer over manual string concatenation for date formatting?

Q5. What is the Unix Epoch and how does JavaScript represent it internally?

CHAPTER SUMMARY

- Dates are internally stored as milliseconds since the Unix Epoch (Jan 1, 1970 UTC).
- Months are 0-indexed (0 = January, 11 = December) — a critical gotcha.
- Use `Intl.DateTimeFormat` for locale-aware formatting and `Intl.RelativeTimeFormat` for relative timestamps.
- Perform arithmetic by adding/subtracting millisecond constants; always compare with `getTime()`.

INTERVIEW CORNER

Q: How do you safely compare two dates in JavaScript? — Convert both to millisecond timestamps with `.getTime()` and compare the numbers. Never use === on Date objects directly because they are reference types and two distinct objects with identical times are never equal by reference.

Q: What is the Temporal API? — The Temporal API is a Stage 3 TC39 proposal designed to replace the `Date` object with immutable, timezone-aware date/time classes (`Temporal.PlainDate`, `Temporal.ZonedDateTime`, etc.) that eliminate most of the footguns in the legacy `Date` API.

15 MODULE

Audio, Video, and the Web Audio API

The web is no longer a text-only medium. Modern browsers ship powerful multimedia engines that allow JavaScript to control audio and video playback, build custom media player interfaces, visualize audio frequencies on Canvas, record microphone input, and even synthesize sounds programmatically. In this module, you will master the **HTMLMediaElement API** for controlling `<audio>` and `<video>` elements, and the **Web Audio API** for advanced audio processing and visualization.

LEARNING OBJECTIVES

- Control `<audio>` and `<video>` elements programmatically with the HTMLMediaElement API
- Understand core properties: `currentTime`, `duration`, `volume`, `playbackRate`, `paused`
- Listen to media events: `loadedmetadata`, `timeupdate`, `play`, `pause`, `ended`, `error`
- Build a custom audio player with play/pause toggle, seek bar, volume slider, and time display
- Use the Web Audio API: `AudioContext`, oscillators, gain nodes, and AnalyserNode
- Visualize audio frequencies using Canvas and `getByteFrequencyData()`
- Explore the Picture-in-Picture API for floating video windows
- Record audio with the MediaRecorder API and `getUserMedia()`

15.1 The HTMLMediaElement API

Both `<audio>` and `<video>` elements share the same JavaScript interface: `HTMLMediaElement`. This API provides properties for inspecting playback state and methods for controlling it:

Table 15.1: Key HTMLMediaElement Properties

Property	Type	Description
`currentTime`	Number	Current playback position in seconds
`duration`	Number	Total length of media in seconds (read-only)
`volume`	Number	Volume level from 0.0 (mute) to 1.0 (max)
`muted`	Boolean	Whether audio output is muted
`paused`	Boolean	Whether playback is currently paused (read-only)
`playbackRate`	Number	Playback speed (1.0 = normal, 2.0 = double)
`ended`	Boolean	Whether playback has reached the end
`readyState`	Number	How much data has been buffered (0–4)

15.2 Media Events

Table 15.2: Essential Media Events

Event	Triggered When
`loadedmetadata`	Duration, video dimensions, and tracks are loaded
`canplay`	Enough data is buffered to begin playback
`play` / `pause`	Playback begins or is halted
`timeupdate`	Current playback position changes (fires 4–5 times/sec)
`waiting`	Playback stopped due to temporary buffer underrun
`ended`	Playback reached the end of the media file
`error`	Media loading or decoding failed
`volumechange`	Volume or muted state changed

15.3 Building a Custom Audio Player

```html
<div class="player-container">
  <audio id="audioPlayer" src="assets/audio/track.mp3" preload="metadata"></audio>

  <div class="player-controls">
    <button id="playPauseBtn" class="btn" aria-label="Play">[>]</button>
    <input type="range" id="seekBar" min="0" max="100" value="0" aria-label="Seek" />
    <span id="timeDisplay">0:00 / 0:00</span>
    <input type="range" id="volumeSlider" min="0" max="1" step="0.05" value="0.8" aria-
    ↪ label="Volume" />
    <select id="speedSelect" aria-label="Playback Speed">
      <option value="0.5">0.5x</option>
      <option value="1" selected>1x</option>
      <option value="1.5">1.5x</option>
      <option value="2">2x</option>
    </select>
  </div>
</div>
```

Code Listing 15.1: Custom Audio Player HTML Structure

```javascript
const audio = document.querySelector("#audioPlayer");
const playPauseBtn = document.querySelector("#playPauseBtn");
const seekBar = document.querySelector("#seekBar");
const timeDisplay = document.querySelector("#timeDisplay");
const volumeSlider = document.querySelector("#volumeSlider");
const speedSelect = document.querySelector("#speedSelect");

// Format seconds into "m:ss" display
function formatTime(seconds) {
  if (isNaN(seconds)) return "0:00";
  const mins = Math.floor(seconds / 60);
  const secs = Math.floor(seconds % 60).toString().padStart(2, "0");
  return `${mins}:${secs}`;
}
```

```
15
16 // Play / Pause Toggle
17 playPauseBtn.addEventListener("click", () => {
18   if (audio.paused) {
19     audio.play();
20     playPauseBtn.textContent = "[||]";
21     playPauseBtn.setAttribute("aria-label", "Pause");
22   } else {
23     audio.pause();
24     playPauseBtn.textContent = "[>]";
25     playPauseBtn.setAttribute("aria-label", "Play");
26   }
27 });
28
29 // Update seek bar and time display as audio plays
30 audio.addEventListener("timeupdate", () => {
31   const progress = (audio.currentTime / audio.duration) * 100;
32   seekBar.value = progress;
33   timeDisplay.textContent = `${formatTime(audio.currentTime)} / ${formatTime(audio.
       ↪ duration)}`;
34 });
35
36 // Seek to a new position when user drags the slider
37 seekBar.addEventListener("input", () => {
38   audio.currentTime = (seekBar.value / 100) * audio.duration;
39 });
40
41 // Volume control
42 volumeSlider.addEventListener("input", () => {
43   audio.volume = parseFloat(volumeSlider.value);
44 });
45
46 // Playback speed control
47 speedSelect.addEventListener("change", () => {
48   audio.playbackRate = parseFloat(speedSelect.value);
49 });
50
51 // Reset UI when track ends
52 audio.addEventListener("ended", () => {
53   playPauseBtn.textContent = "[>]";
54   seekBar.value = 0;
55 });
```

Code Listing 15.2: Custom Audio Player JavaScript Controller

15.4 The Web Audio API: Synthesis and Analysis

The **Web Audio API** provides a powerful, low-level audio processing graph. Audio flows from **source nodes** through **processing nodes** (gain, filters, effects) to the **destination** (speakers):

Figure 15.1: Web Audio API Signal Processing Graph

```
// Create an AudioContext (the main entry point)
const audioCtx = new AudioContext();

function playTone(frequency = 440, duration = 0.5) {
  // 1. Source: Create an oscillator (sine wave generator)
  const oscillator = audioCtx.createOscillator();
  oscillator.type = "sine"; // "sine", "square", "sawtooth", "triangle"
  oscillator.frequency.setValueAtTime(frequency, audioCtx.currentTime);

  // 2. Processing: Create a gain node for volume envelope
  const gainNode = audioCtx.createGain();
  gainNode.gain.setValueAtTime(0.3, audioCtx.currentTime);
  // Fade out to prevent audio clicks:
  gainNode.gain.exponentialRampToValueAtTime(0.001, audioCtx.currentTime + duration);

  // 3. Connect the audio graph: Oscillator -> Gain -> Speakers
  oscillator.connect(gainNode);
  gainNode.connect(audioCtx.destination);

  // 4. Start and schedule automatic stop
  oscillator.start();
  oscillator.stop(audioCtx.currentTime + duration);
}

// Play a musical note (A4 = 440Hz)
playTone(440, 1.0);
```

Code Listing 15.3: Synthesizing a Tone with Web Audio API

15.5 Audio Frequency Visualization with Canvas

The `AnalyserNode` performs Fast Fourier Transform (FFT) analysis on audio data, providing frequency-domain data that can be rendered as bar charts or waveforms:

```
const canvas = document.querySelector("#visualizerCanvas");
const canvasCtx = canvas.getContext("2d");

// Create an AnalyserNode
const analyser = audioCtx.createAnalyser();
analyser.fftSize = 256; // Must be a power of 2
const bufferLength = analyser.frequencyBinCount; // fftSize / 2
const dataArray = new Uint8Array(bufferLength);

// Connect audio source to analyser
// sourceNode.connect(analyser);
// analyser.connect(audioCtx.destination);

function drawBars() {
  requestAnimationFrame(drawBars);
  analyser.getByteFrequencyData(dataArray); // Fill array with frequency magnitudes

  canvasCtx.fillStyle = "#1A1A1A";
  canvasCtx.fillRect(0, 0, canvas.width, canvas.height);
```

```

  const barWidth = (canvas.width / bufferLength) * 2.5;
  let x = 0;

  for (let i = 0; i < bufferLength; i++) {
    const barHeight = dataArray[i]; // 0-255 intensity
    const hue = (i / bufferLength) * 120 + 30; // Gold-to-green gradient
    canvasCtx.fillStyle = `hsl(${hue}, 80%, 50%)`;
    canvasCtx.fillRect(x, canvas.height - barHeight, barWidth, barHeight);
    x += barWidth + 1;
  }
}

drawBars(); // Start animation loop
```

Code Listing 15.4: Real-Time Frequency Bar Visualizer

15.6 Picture-in-Picture API

The Picture-in-Picture (PiP) API allows a video element to float in a small overlay window above other application windows:

```
const video = document.querySelector("#videoPlayer");
const pipBtn = document.querySelector("#pipButton");

pipBtn.addEventListener("click", async () => {
  if (document.pictureInPictureElement) {
    await document.exitPictureInPicture();
  } else if (document.pictureInPictureEnabled) {
    await video.requestPictureInPicture();
  }
});

video.addEventListener("enterpictureinpicture", () => {
  pipBtn.textContent = "Exit PiP";
});

video.addEventListener("leavepictureinpicture", () => {
  pipBtn.textContent = "Enter PiP";
});
```

Code Listing 15.5: Activating Picture-in-Picture Mode

15.7 Recording Audio with MediaRecorder

```
let mediaRecorder;
let audioChunks = [];

async function startRecording() {
```

```
  const stream = await navigator.mediaDevices.getUserMedia({ audio: true });
  mediaRecorder = new MediaRecorder(stream);

  mediaRecorder.addEventListener("dataavailable", (e) => {
    audioChunks.push(e.data);
  });

  mediaRecorder.addEventListener("stop", () => {
    const audioBlob = new Blob(audioChunks, { type: "audio/webm" });
    const audioUrl = URL.createObjectURL(audioBlob);
    const playback = new Audio(audioUrl);
    playback.play();
    audioChunks = []; // Reset for next recording
  });

  mediaRecorder.start();
  console.log("Recording started...");
}

function stopRecording() {
  mediaRecorder.stop();
  console.log("Recording stopped.");
}
```

Code Listing 15.6: Recording Microphone Audio

MINI CHALLENGE

The Piano Keyboard Synthesizer. Build an interactive piano keyboard with 12 keys (one octave) using the Web Audio API. Each key should play its corresponding musical frequency when clicked or when the corresponding keyboard letter is pressed. Use the formula: $f = 440 \times 2^{(n-49)/12}$ where n is the key number (A4 = key 49).

15.8 Exercises

E1. [Level 1] Display the total duration of a video file formatted as `"mm:ss"` as soon as `loadedmetadata` fires.
E2. [Level 1] Build a play/pause toggle button that updates its icon based on the `paused` state.
E3. [Level 2] Build a custom seek bar that visually tracks playback progress using `timeupdate`.
E4. [Level 2] Implement a playback speed controller with options for 0.5x, 1x, 1.5x, and 2x.
E5. [Level 3] Build a frequency bar visualizer using Canvas and the Web Audio API `AnalyserNode`.
E6. [Level 4] Implement a voice recorder using `getUserMedia` and `MediaRecorder` that saves recordings as downloadable `.webm` files.

15.9 Quiz

Q1. What event fires approximately 4–5 times per second during audio/video playback?
Q2. How do you reset an audio element to replay from the beginning without reloading?
Q3. What is the purpose of the `AnalyserNode` in the Web Audio API?
Q4. What does `audio.playbackRate = 2.0` do?

Q5. Why should you use `gainNode.gain.exponentialRampToValueAtTime()` before stopping an oscillator?

CHAPTER SUMMARY

- The HTMLMediaElement API controls `<audio>` and `<video>` playback, seeking, volume, and speed.
- The Web Audio API provides low-level audio synthesis, processing, and FFT analysis.
- Use `AnalyserNode` with Canvas to build real-time audio frequency visualizations.
- The MediaRecorder API captures microphone/camera streams for recording applications.

INTERVIEW CORNER

Q: How does the Web Audio API differ from the HTMLMediaElement API? — The HTMLMediaElement API is a high-level interface for playing audio/video files with basic controls (play, pause, seek, volume). The Web Audio API is a low-level, graph-based audio processing engine capable of synthesizing sounds, applying real-time effects (reverb, compression, filtering), performing FFT frequency analysis, and routing audio through complex signal chains.

Q: Why does `audio.play()` return a Promise? — Because modern browsers block autoplay by default until the user interacts with the page. The returned Promise rejects with a `NotAllowedError` if autoplay was blocked, allowing you to catch and handle the restriction gracefully.

16 MODULE

Maps and Sets

JavaScript developers often reach for plain objects when they need key-value storage, and arrays when they need collections of unique values. While these work for simple cases, they have fundamental limitations: objects coerce all keys to strings, lack a built-in size property, and carry inherited prototype properties. ES6 introduced two purpose-built collection types — **Map** and **Set** — designed to solve these exact problems with guaranteed performance and clean APIs.

LEARNING OBJECTIVES

- Create and manipulate `Map` collections with any key type (objects, functions, numbers)
- Understand the performance advantages of `Map` over plain objects for dynamic key-value storage
- Create and manipulate `Set` collections for guaranteed-unique value storage
- Perform set-theory operations: union, intersection, difference, and symmetric difference
- Understand `WeakMap` and `WeakSet` for memory-safe object associations
- Iterate Maps and Sets with `for...of`, `forEach`, destructuring, and spread syntax
- Choose the correct data structure for each use case based on performance characteristics

16.1 The Map Data Structure

A `Map` is an ordered collection of key-value pairs where keys can be **any data type** — objects, functions, numbers, booleans, even `NaN`:

```
const userPermissions = new Map();

// Setting key-value pairs
userPermissions.set("fedi", { role: "admin", canDeploy: true });
userPermissions.set("sam", { role: "editor", canDeploy: false });
userPermissions.set("alex", { role: "viewer", canDeploy: false });

// Getting values
console.log(userPermissions.get("fedi")); // { role: "admin", canDeploy: true }

// Checking existence
console.log(userPermissions.has("sam")); // true
console.log(userPermissions.has("unknown")); // false

// Size property (no need for Object.keys().length!)
console.log(userPermissions.size); // 3

// Deleting entries
userPermissions.delete("alex");
```

```
console.log(userPermissions.size); // 2
```

Code Listing 16.1: Creating and Using Maps

16.2 Map vs Plain Object: When to Use Which

Table 16.1: Map vs Plain Object Feature Comparison

Feature	Plain Object	Map
Key Types	Strings and Symbols only	Any type (Objects, Numbers, etc.)
Key Order	Complex historical rules	Strict insertion order guaranteed
Size Property	Manual `Object.keys().length`	Built-in `map.size`
Iteration	Via `Object.entries()`	Directly iterable with `for...of`
Performance	Optimized for small records	Optimized for frequent add/delete
Prototype Pollution	Inherits from `Object.prototype`	No inherited keys
Serialization	Native `JSON.stringify()`	Requires manual conversion

```
// PROBLEM: Object coerces ALL keys to strings!
const objCache = {};
objCache[1] = "one";
objCache["1"] = "string-one";
console.log(Object.keys(objCache)); // ["1"] (only one key! number was coerced)

// SOLUTION: Map preserves key identity
const mapCache = new Map();
mapCache.set(1, "one");         // Key is number 1
mapCache.set("1", "string-one"); // Key is string "1"
console.log(mapCache.size);      // 2 (both keys preserved!)
```

Code Listing 16.2: Object Keys are Always Strings

BEST PRACTICE

Use a **plain object** for static configuration records, JSON-serializable data, and objects with a fixed, known set of properties. Use a **Map** when keys are dynamic, when you need non-string keys, when insertion order matters, or when you frequently add and delete entries.

16.3 Object Keys with Maps

```
const elementMetadata = new Map();

const btn = document.querySelector("#submitBtn");
const form = document.querySelector("#mainForm");

// DOM elements as keys!
elementMetadata.set(btn, { clickCount: 0, lastClicked: null });
elementMetadata.set(form, { submissionCount: 0 });
```

```

btn.addEventListener("click", () => {
  const meta = elementMetadata.get(btn);
  meta.clickCount++;
  meta.lastClicked = new Date();
  console.log(`Button clicked ${meta.clickCount} times`);
});
```

Code Listing 16.3: Using Objects and DOM Elements as Map Keys

16.4 Iterating Maps

```
const inventory = new Map([
  ["Yamaha MT-07", 8199],
  ["Kawasaki Ninja ZX-6R", 11999],
  ["BMW R 1250 GS", 17995]
]);

// 1. for...of with destructuring (preferred)
for (const [model, price] of inventory) {
  console.log(`${model}: $${price.toLocaleString()}`);
}

// 2. forEach callback
inventory.forEach((price, model) => {
  console.log(`${model} costs $${price}`);
});

// 3. Convert to array of entries
const sorted = [...inventory].sort((a, b) => a[1] - b[1]);
console.log(sorted[0]); // ["Yamaha MT-07", 8199]
```

Code Listing 16.4: Map Iteration Patterns

16.5 The Set Data Structure

A `Set` is a collection of **unique values**. Adding a duplicate value is silently ignored. Sets use the SameValueZero algorithm for equality, meaning `NaN === NaN` evaluates as `true` inside a Set:

```
const uniqueTags = new Set(["javascript", "web", "javascript", "css", "web"]);
console.log(uniqueTags.size); // 3 (duplicates removed!)
console.log(uniqueTags); // Set {"javascript", "web", "css"}

// Adding and deleting
uniqueTags.add("react");
uniqueTags.delete("css");

// Checking membership (O(1) average time!)
console.log(uniqueTags.has("javascript")); // true
console.log(uniqueTags.has("python"));     // false

```

```
// Array deduplication in one line
const numbers = [1, 2, 3, 2, 1, 4, 5, 4];
const unique = [...new Set(numbers)];
console.log(unique); // [1, 2, 3, 4, 5]
```

Code Listing 16.5: Creating and Using Sets

16.6 Set-Theory Operations

ES2025 added native Set methods for mathematical set operations. For older environments, implement them manually:

```
const frontendSkills = new Set(["HTML", "CSS", "JavaScript", "React"]);
const backendSkills = new Set(["Node.js", "JavaScript", "PostgreSQL", "Docker"]);

// UNION: All unique skills from both sets
const allSkills = new Set([...frontendSkills, ...backendSkills]);
console.log(allSkills);
// Set {"HTML", "CSS", "JavaScript", "React", "Node.js", "PostgreSQL", "Docker"}

// INTERSECTION: Skills common to both sets
const sharedSkills = new Set(
  [...frontendSkills].filter(skill => backendSkills.has(skill))
);
console.log(sharedSkills); // Set {"JavaScript"}

// DIFFERENCE: Skills in frontend but NOT in backend
const frontendOnly = new Set(
  [...frontendSkills].filter(skill => !backendSkills.has(skill))
);
console.log(frontendOnly); // Set {"HTML", "CSS", "React"}

// SYMMETRIC DIFFERENCE: Skills in either set but NOT in both
const exclusive = new Set(
  [...allSkills].filter(skill => !sharedSkills.has(skill))
);
console.log(exclusive);
// Set {"HTML", "CSS", "React", "Node.js", "PostgreSQL", "Docker"}
```

Code Listing 16.6: Set-Theory Operations in JavaScript

16.7 WeakMap and WeakSet: Memory-Safe References

A `WeakMap` holds **weak references** to its object keys. When no other references to a key object exist, the garbage collector automatically reclaims both the key and its associated value — preventing memory leaks:

```
// Pattern: Private data storage without polluting the object
const privateData = new WeakMap();
```

```

class User {
  constructor(name, password) {
    this.name = name;
    // Store sensitive data in WeakMap (not visible on the object!)
    privateData.set(this, { password, loginAttempts: 0 });
  }

  authenticate(attempt) {
    const data = privateData.get(this);
    data.loginAttempts++;
    return attempt === data.password;
  }
}

const fedi = new User("Fedi", "securePass123");
console.log(fedi.authenticate("wrong"));      // false
console.log(fedi.authenticate("securePass123")); // true
console.log(fedi.password); // undefined (private data is hidden!)
```

Code Listing 16.7: WeakMap for Private Data and Caching

NOTE

WeakMap keys must be objects (not primitives). WeakMaps are **not iterable** and have no `size` property because their contents can be garbage-collected at any time. Use them for associating metadata with objects without preventing cleanup.

MINI CHALLENGE

The Tag Cloud Generator. Given an array of blog post objects, each with a `tags` array property, use a `Map` to count the frequency of each tag across all posts. Then sort the entries by frequency in descending order and display the top 10 most popular tags.

16.8 Exercises

E1. [Level 1] Create a `Map` storing 5 country-capital pairs and iterate it with `for...of`.
E2. [Level 1] Deduplicate the array `[1, 2, 3, 2, 1, 4, 5, 4]` using a `Set` in one line.
E3. [Level 2] Write a function `wordFrequency(text)` using a `Map` to count occurrences of each word.
E4. [Level 2] Implement `setIntersection(setA, setB)` and `setDifference(setA, setB)` utility functions.
E5. [Level 3] Use a `WeakMap` to cache expensive computation results keyed by input objects, allowing garbage collection of unused cache entries.
E6. [Level 4] Build an LRU (Least Recently Used) Cache class using a `Map` that maintains insertion order and evicts the oldest entry when capacity is exceeded.

16.9 Quiz

Q1. Why can a `Map` use objects as keys but a plain object cannot?
Q2. What is the time complexity of `Set.has()` compared to `Array.includes()`?

Q3. Why are WeakMaps not iterable?
Q4. What happens when you add a duplicate value to a `Set`?
Q5. When should you prefer a plain object over a `Map`?

CHAPTER SUMMARY

- `Map` stores ordered key-value pairs with any key type and guaranteed $O(1)$ average lookups.
- `Set` stores unique values with $O(1)$ membership testing, replacing manual deduplication loops.
- `WeakMap` and `WeakSet` hold weak object references that allow automatic garbage collection.
- Use spread and filter to implement set-theory operations (union, intersection, difference).

INTERVIEW CORNER

Q: What is the time complexity of Map.get() vs Object property access? — Both are $O(1)$ average-case. However, Map is optimized for frequent additions and deletions, while plain objects are optimized for fixed-shape access patterns. V8 creates hidden classes for objects with stable property shapes, making repeated property access extremely fast.

Q: How would you implement an LRU Cache in JavaScript? — Use a `Map` because it maintains insertion order. On `get(key)`, delete and re-insert the entry to move it to the end (most recently used). On `set(key, value)`, if capacity is exceeded, delete `map.keys().next().value` (the oldest/least recently used entry). This gives $O(1)$ get and set operations.

17 MODULE

localStorage and CRUD Operations

Until now, whenever you refreshed your browser page, all variables and application state vanished. To make a web application genuinely useful, data must persist across page refreshes and browser restarts. The **Web Storage API** (`localStorage`) provides a simple, synchronous, key-value storage engine built directly into the browser. In this module, you will master client-side persistence, navigate serialization limits, handle quota boundaries, and implement the four fundamental database operations: **Create, Read, Update, and Delete (CRUD)**.

LEARNING OBJECTIVES

- Understand client storage options: `localStorage`, `sessionStorage`, Cookies, and IndexedDB
- Master the Web Storage API: `setItem`, `getItem`, `removeItem`, and `clear`
- Serialize and deserialize arrays and objects, and build custom JSON replacers/revivers for Dates, Maps, and Sets
- Handle storage quota overflow errors (`QuotaExceededError`) and write eviction routines
- Build a complete, stateful CRUD data management service class
- Listen to multi-tab storage synchronization using the `storage` event
- Review security implications, XSS risks, and token storage best practices

17.1 The Browser Storage Landscape

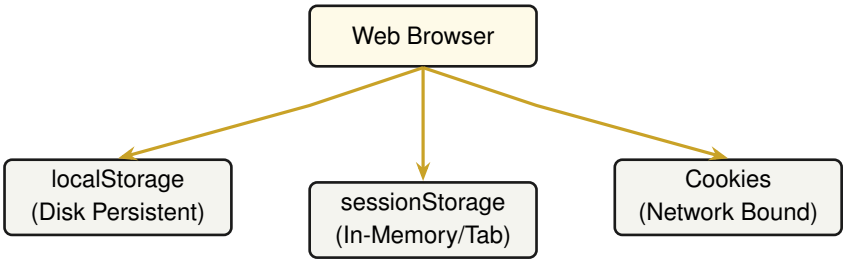
Modern browsers provide several ways to store data locally:

Table 17.1: Comparison of Client-Side Storage Mechanisms

Storage	Capacity	Lifetime	Accessible by	Server Sent?
`localStorage`	~5MB	Indefinite (until cleared)	Same Origin	No
`sessionStorage`	~5MB	Tab/Window session	Same Tab Only	No
`Cookies`	~4KB	Configurable expiration	Origin	Yes (with requests)
`IndexedDB`	Hundreds MB+	Indefinite	Same Origin	No (Async)

17.2 The localStorage API

The Web Storage API exposes a simple key-value interface. Note that `localStorage` stores **strings only**. Attempting to store an object, number, or boolean directly will silently coerce it to a string representation:



All storage is partitioned by **Origin** (Protocol + Domain + Port).
HTTPS and HTTP sites have completely isolated storage keys.

Figure 17.1: Browser Client-Side Storage Architecture

```
// 1. Saving a string item
localStorage.setItem("fedi_theme", "dark");

// 2. Reading an item (returns null if key does not exist)
const currentTheme = localStorage.getItem("fedi_theme");
console.log(currentTheme); // "dark"

// 3. Removing an item
localStorage.removeItem("fedi_theme");

// 4. Clearing all keys for current origin
// localStorage.clear();
```

Code Listing 17.1: Basic localStorage operations

17.3 Serialization Boundaries and Custom JSON Replacers

While `JSON.stringify()` and `JSON.parse()` work well for simple objects, they fail for native structures like `Date`, `Map`, and `Set`:

- `Date` is serialized to an ISO string, but parsed back as a string, not a Date.
- `Map` and `Set` serialize to empty object literals (`{}`) or arrays.

To solve this, implement custom JSON replacers and revivers:

```
const state = {
  createdAt: new Date("2026-08-16T12:00:00Z"),
  roles: new Set(["admin", "developer"]),
  cache: new Map([["key1", "val1"]])
};

// Custom Replacer function to convert Set and Map into arrays
function replacer(key, value) {
  if (value instanceof Set) {
    return { __dataType: "Set", value: Array.from(value) };
  }
  if (value instanceof Map) {
    return { __dataType: "Map", value: Array.from(value.entries()) };
```

```
  }
  return value;
}

// Custom Reviver function to reconstruct original types
function reviver(key, value) {
  if (value && typeof value === "object") {
    if (value.__dataType === "Set") return new Set(value.value);
    if (value.__dataType === "Map") return new Map(value.value);
  }
  // Convert ISO date strings back to real Date objects
  const datePattern = /^\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}/;
  if (typeof value === "string" && datePattern.test(value)) {
    return new Date(value);
  }
  return value;
}

const serialized = JSON.stringify(state, replacer);
localStorage.setItem("fedi_state", serialized);

const loadedState = JSON.parse(localStorage.getItem("fedi_state"), reviver);
console.log(loadedState.createdAt instanceof Date); // true!
console.log(loadedState.roles instanceof Set);        // true!
console.log(loadedState.cache.get("key1"));          // "val1"
```

Code Listing 17.2: Custom Replacer and Reviver for Complex Storage

17.4 Quota Limits and QuotaExceededError

Most browsers enforce a quota of **5MB** per origin. Attempting to write beyond this quota will throw a fatal `QuotaExceededError` exception. Always write fallback structures when saving payload strings:

```
function safeStore(key, dataString) {
  try {
    localStorage.setItem(key, dataString);
  } catch (error) {
    if (error.name === "QuotaExceededError" || error.code === 22) {
      console.warn("Storage quota exceeded! Wiping cache and retrying...");
      // Eviction Strategy: Clear old non-critical caches
      clearNonCriticalCaches();
      try {
        localStorage.setItem(key, dataString); // Retry once
      } catch (retryError) {
        console.error("Second storage attempt failed:", retryError.message);
      }
    } else {
      throw error; // Propagate other errors
    }
  }
}

```

```
function clearNonCriticalCaches() {
  // Loop through keys and evict temporary cache entries
  for (let i = localStorage.length - 1; i >= 0; i--) {
    const key = localStorage.key(i);
    if (key.startsWith("temp_cache_")) {
      localStorage.removeItem(key);
    }
  }
}
```

Code Listing 17.3: Handling Storage Quota Failures Gracefully

17.5 Implementing the CRUD Cycle

Let us build a full, reusable CRUD persistence service class:

```
class TaskRepository {
  constructor(storageKey = "fedi_tasks_db") {
    this.storageKey = storageKey;
  }

  // 1. READ ALL
  getAll() {
    try {
      const raw = localStorage.getItem(this.storageKey);
      return raw ? JSON.parse(raw) : [];
    } catch (e) {
      console.error("Failed to read from localStorage:", e);
      return [];
    }
  }

  // Helper: SAVE ALL
  _save(tasks) {
    localStorage.setItem(this.storageKey, JSON.stringify(tasks));
  }

  // 2. CREATE
  create(title) {
    const tasks = this.getAll();
    const newTask = {
      id: "TASK-" + Date.now(),
      title: title.trim(),
      completed: false,
      createdAt: new Date().toISOString()
    };
    tasks.push(newTask);
    this._save(tasks);
    return newTask;
  }

  // 3. UPDATE
  update(id, updates) {
```

```
    const tasks = this.getAll();
    const index = tasks.findIndex(t => t.id === id);
    if (index === -1) return null;

    tasks[index] = { ...tasks[index], ...updates };
    this._save(tasks);
    return tasks[index];
  }

  // 4. DELETE
  delete(id) {
    const tasks = this.getAll();
    const filtered = tasks.filter(t => t.id !== id);
    this._save(filtered);
    return filtered.length !== tasks.length;
  }
}
```

Code Listing 17.4: A Complete CRUD Task Repository Service

17.6 Cross-Tab State Synchronization

When a user modifies `localStorage` in one browser tab, other open tabs of the same website receive a `storage` event. This is useful for synchronizing UI state across tabs:

```
window.addEventListener("storage", (event) => {
  // Event properties: key, oldValue, newValue, url, storageArea
  if (event.key === "fedi_tasks_db") {
    console.log("Tasks modified in another tab! Updating UI...");
    const updatedTasks = JSON.parse(event.newValue);
    renderTaskList(updatedTasks);
  }
});
```

Code Listing 17.5: Listening to changes across multiple tabs

BEST PRACTICE

`localStorage` is synchronous and blocking. Reading or writing multi-megabyte JSON strings will freeze UI animations and click handlers. For datasets exceeding a few megabytes, use the asynchronous **IndexedDB API** (or library wrappers like `localForage`).

MINI CHALLENGE

The Persistent Theme Switcher. Create a toggle button that switches the webpage between `"light"` and `"dark"` theme. Persist the chosen theme in `localStorage`. When the page is reloaded, read the persisted theme and apply it immediately before rendering the body to prevent any theme flickering.

17.7 Exercises

E1. [Level 1] Write helper functions `saveString(key, val)` and `getString(key, defaultVal)`.
E2. [Level 1] Write code that tracks how many times a user has visited your webpage by incrementing a counter in `localStorage`.
E3. [Level 2] Build a function `clearExpiredData(key)` where objects are saved with a timestamp and TTL (time-to-live) and wiped if expired.
E4. [Level 2] Calculate the total storage space currently occupied by your application's `localStorage` keys in kilobytes.
E5. [Level 3] Build an offline-capable draft autosave feature for a blog post form that saves user typing every 2 seconds.
E6. [Level 4] Build a generic `LocalStorageCollection<T>` class that provides indexing, pagination (`getPage(page, size)`), and query filtering.

17.8 Quiz

Q1. What data type does `localStorage.getItem()` return if the key does not exist?
Q2. What happens if you store an object directly into `localStorage` without `JSON.stringify`?
Q3. What is the standard storage limit for `localStorage` per origin in modern browsers?
Q4. When is the `window` `storage` event triggered?
Q5. Why should you never store sensitive secrets like JWT authentication tokens or credit card numbers in `localStorage`?

CHAPTER SUMMARY

- `localStorage` provides persistent, origin-scoped key-value storage across browser sessions.
- Always serialize complex structures with `JSON.stringify` and parse with `JSON.parse` inside a `try...catch`.
- Implement full CRUD operations: Create (`push + setItem`), Read (`getItem + parse`), Update (`map` / index replace), Delete (`filter`).
- Use the `storage` event to synchronize application state across multiple browser tabs in real time.

INTERVIEW CORNER

Q: Why is localStorage vulnerable to XSS (Cross-Site Scripting)? — Any JavaScript code executing on the page (including third-party analytics scripts or injected malicious scripts) has full read/write access to `localStorage`. For sensitive auth tokens, `HttpOnly` cookies are preferred because they cannot be accessed by client-side JavaScript.

Q: How does localStorage differ from sessionStorage? — Both share the same API interface, capacity, and origin restrictions. However, `localStorage` persists indefinitely until cleared, whereas `sessionStorage` is cleared as soon as the tab or window is closed. Additionally, `sessionStorage` data is not shared between different tabs, even if they have the same origin.

PART III

Tier 3 — Advanced

TIER 3 — ADVANCED

Here the craft deepens. Classes give you professional object design. Recursion, data structures, and algorithms teach you to reason about correctness and performance — the skills that separate programmers from coders.

18 MODULE

Classes and Object-Oriented Programming

As applications scale from small scripts into enterprise software systems, organizing code into well-structured, modular entities becomes essential. **Object-Oriented Programming (OOP)** is a paradigm centered around objects that encapsulate data and behaviors. Introduced in ES6, JavaScript's **Class** syntax provides a clean, modern, standard way to build object hierarchies, enforce encapsulation with private fields, and share functionality through inheritance. In this module, we will cover advanced class patterns, explore the prototype compile target, prevent abstract class instantiations, and implement multi-object mixin composition.

LEARNING OBJECTIVES

- Understand JavaScript class syntax, constructors, and instance instantiations
- Encapsulate internal state with modern private fields (`#privateField`) and the `in` check operator
- Implement class inheritance with `extends` and the `super` keyword
- Use the `new.target` meta-property to build abstract parent classes
- Define utility logic with `static` methods, properties, and private static fields
- Protect and validate property mutations using getters and setters
- Implement Mixin composition to achieve multiple-inheritance patterns
- Understand memory optimization: constructor properties vs prototype methods
- Leverage `Symbol.iterator` to build custom iterable classes

18.1 Introduction to Classes

A class is a blueprint for creating objects. You define a class with the `class` keyword and instantiate new objects using the `new` keyword:

```
class User {
  // The constructor initializes instance properties
  constructor(name, email) {
    this.name = name;
    this.email = email;
    this.createdAt = new Date();
  }

  // Instance method (shared by all instances via prototype)
  sendWelcomeEmail() {
    console.log(`Sending welcome email to ${this.name} <${this.email}>...`);
```

```
  }
}

// Instantiating objects
const userA = new User("Fedi", "fedi@fediapp.dev");
userA.sendWelcomeEmail(); // "Sending welcome email to Fedi <fedi@fediapp.dev>..."
```

Code Listing 18.1: Defining and instantiating a class

18.2 Private Fields, Encapsulation, and the "in" Operator

Modern JavaScript (ES2022+) provides true language-level private fields prefixed with a hash (#). Private fields cannot be accessed or mutated outside the class body. You can check if an object has a private field using the `in` operator:

```
class BankAccount {
  // Declaring private fields
  #balanceCents;
  #accountPin;

  constructor(owner, initialDepositCents, pin) {
    this.owner = owner;
    this.#balanceCents = initialDepositCents;
    this.#accountPin = pin;
  }

  deposit(cents) {
    if (cents <= 0) throw new Error("Deposit amount must be positive");
    this.#balanceCents += cents;
    return this.#balanceCents;
  }

  withdraw(cents, enteredPin) {
    if (enteredPin !== this.#accountPin) throw new Error("Invalid PIN!");
    if (cents > this.#balanceCents) throw new Error("Insufficient funds!");
    this.#balanceCents -= cents;
    return this.#balanceCents;
  }

  getBalance(enteredPin) {
    if (enteredPin !== this.#accountPin) throw new Error("Access Denied!");
    return `$${(this.#balanceCents / 100).toFixed(2)}`;
  }

  // Static checker using 'in' operator to verify private field presence
  static isBankAccount(obj) {
    return #balanceCents in obj;
  }
}

const fediVault = new BankAccount("Fedi", 500000, 1234);
console.log(BankAccount.isBankAccount(fediVault)); // true
```

```
console.log(BankAccount.isBankAccount({}));         // false
```

Code Listing 18.2: Enforcing encapsulation with private fields

18.3 Getters, Setters, and Private Statics

Getters and setters allow you to intercept property access and assignments. Classes also support private static fields for sharing hidden configurations across all instances:

```
class TemperatureSensor {
  #celsius;
  static #MAX_LIMIT = 100; // Private static constant

  constructor(celsius) {
    this.celsius = celsius; // Invokes setter
  }

  get celsius() {
    return this.#celsius;
  }

  set celsius(value) {
    if (value < -273.15) throw new RangeError("Below absolute zero!");
    if (value > TemperatureSensor.#MAX_LIMIT) {
      throw new RangeError("Exceeds safe limits!");
    }
    this.#celsius = value;
  }

  get fahrenheit() {
    return (this.#celsius * 9/5) + 32;
  }
}
```

Code Listing 18.3: Using Getters, Setters, and Private Statics

18.4 Inheritance and Abstract Classes

Inheritance allows a child class to inherit properties and methods from a parent class. JavaScript does not have a native `abstract` keyword, but you can build abstract classes (which cannot be instantiated directly, only extended) using the `new.target` meta-property:

```
class AbstractComponent {
  constructor(elementId) {
    if (new.target === AbstractComponent) {
      throw new TypeError("Cannot instantiate abstract class AbstractComponent directly.")
    ↪ ;
    }
    this.elementId = elementId;
```

```
  }

  // Abstract method placeholder
  render() {
    throw new Error("Method 'render()' must be implemented by child classes.");
  }
}

class ButtonComponent extends AbstractComponent {
  constructor(elementId, label) {
    super(elementId);
    this.label = label;
  }

  render() {
    return `<button id="${this.elementId}">${this.label}</button>`;
  }
}

// const comp = new AbstractComponent("main"); // Throws TypeError!
const btn = new ButtonComponent("btn1", "Submit"); // Works!
```

Code Listing 18.4: Abstract Classes with new.target

18.5 Class Mixins and Multiple Inheritance

JavaScript classes support single-parent inheritance only. To share reusable behaviors across unrelated classes, implement the **Mixin Pattern** using function-wrapped classes:

```
// Mixin for Logging behavior
const Loggable = (BaseClass) => class extends BaseClass {
  log(message) {
    console.log(`[${new Date().toISOString()}] ${message}`);
  }
};

// Mixin for JSON Serialization
const Serializable = (BaseClass) => class extends BaseClass {
  serialize() {
    return JSON.stringify(this);
  }
};

class BaseUser {
  constructor(name) { this.name = name; }
}

// Combine mixins into a composite inheritance chain
class SmartUser extends Loggable(Serializable(BaseUser)) {
  constructor(name, role) {
    super(name);
    this.role = role;
  }
```

```
}

const fedi = new SmartUser("Fedi", "admin");
fedi.log("User session started."); // Outputs log prefix
console.log(fedi.serialize());       // {"name":"Fedi","role":"admin"}
```

Code Listing 18.5: Implementing Mixins in JavaScript

18.6 Prototypal Inheritance and Memory Optimization

JavaScript classes are syntactic sugar over prototypal delegation. To optimize memory, methods defined on the class body are attached to the `prototype` object, meaning they are shared across all instances. Properties assigned inside the constructor are duplicated for every single instantiated object.

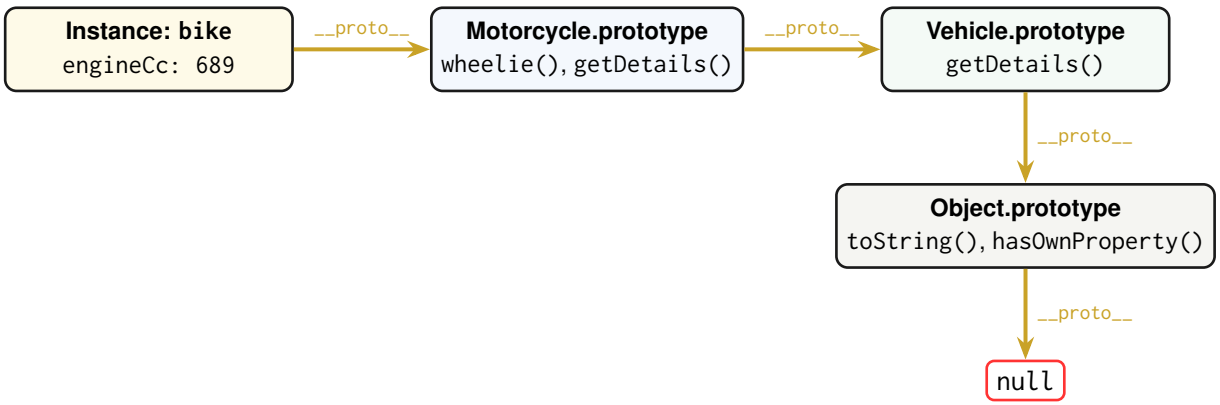


Figure 18.1: The Prototype Chain Lookup Mechanism

```
class TaskList {
  constructor() {
    this.tasks = [];
  }
  addTask(task) { this.tasks.push(task); }

  // Implementing iterator protocol to support for...of directly
  *[Symbol.iterator]() {
    for (const task of this.tasks) {
      yield task;
    }
  }
}

const list = new TaskList();
list.addTask("Design Book Cover");
list.addTask("Review Code Snippets");

for (const task of list) {
  console.log(task); // Iterates list items directly!
}
```

Code Listing 18.6: Custom Iterables with Symbol.iterator

MINI CHALLENGE

The Observable Model Class. Implement an abstract `EventEmitter` class with `on(event, listener)`, `emit(event, data)`, and `off(event, listener)` methods. Then create a `StoreModel` class that extends `EventEmitter` and emits a `"stateChanged"` event whenever its internal state is updated.

18.7 Exercises

E1. [Level 1] Create a `Book` class with `title`, `author`, and `isbn`. Add a method `getSummary()`.
E2. [Level 1] Create a `Counter` class with private field `#count` and methods `increment()`, `decrement()`, and getter `count`.
E3. [Level 2] Create an `Employee` parent class and a `Developer` child class that adds a `programmingLanguages` array.
E4. [Level 2] Implement a `MathUtils` class with private constructor and static methods for `clamp`, `lerp`, and `roundTo`.
E5. [Level 3] Build a complete `LinkedList` or `Queue` data structure as an ES6 class.
E6. [Level 4] Implement a complete State Machine class hierarchy with dynamic state transitions, guard conditions, and transition callbacks.

18.8 Quiz

Q1. What must you call first in a child constructor before using `this`?
Q2. How are private class fields denoted in modern JavaScript syntax?
Q3. What is the difference between an instance method and a static method?
Q4. What is the prototype chain and how does property resolution travel along it?
Q5. What does `instanceof` check?

CHAPTER SUMMARY

- Classes provide clean object blueprints with constructors, methods, and getters/setters.
- Enforce true encapsulation using `#privateField` and `#privateMethod`.
- Share behavior with `class Child extends Parent` and invoke the parent constructor with `super()`.
- Static methods and fields are attached directly to the constructor class function.
- Classes are syntactic sugar over JavaScript's underlying prototypal inheritance chain.

INTERVIEW CORNER

Q: How does JavaScript's prototypal inheritance differ from classical inheritance? — In classical languages (Java/C++), classes are distinct compile-time types that create instance copies. In JavaScript, inheritance is dynamic delegation: objects link directly to prototype objects via the prototype chain (`[[Prototype]]`). Property lookups travel up the chain until found or `null` is reached.

Q: Why are methods defined inside the class body more memory-efficient? — Methods defined in the class body are attached once to the constructor's `prototype` property. All instances reference these same functions via delegation. If you define methods in the constructor using `this.method = function()`, each instantiated object creates its own separate closure function

in memory, causing significant waste.

19 MODULE

Recursion

Recursion is one of the most powerful and elegant techniques in computer science. A recursive function calls **itself** to solve progressively smaller instances of the same problem until it reaches a trivially solvable **base case**. Recursion is the natural way to process hierarchical data structures (trees, nested objects, file systems), implement divide-and-conquer algorithms (Merge Sort, Quick Sort), and solve mathematical recurrences (factorials, Fibonacci, permutations).

LEARNING OBJECTIVES

- Understand the anatomy of recursion: base case, recursive step, and convergence
- Trace recursive function calls through the Call Stack frame by frame
- Implement classic recursive algorithms: factorial, Fibonacci, power, sum of digits
- Process nested/hierarchical data: deep object traversal, DOM tree walking, flatten arrays
- Understand and prevent stack overflow errors with clear base cases
- Optimize recursive functions with tail call optimization (TCO) concepts
- Compare recursion vs iteration: readability, performance, and stack depth tradeoffs

19.1 The Anatomy of Recursion

Every recursive function must contain exactly two components:

1. **Base Case**: The condition that **stops** the recursion. Without it, the function calls itself infinitely until the Call Stack overflows.
2. **Recursive Step**: The function calls itself with a **smaller** or **simpler** input, converging toward the base case with every call.

```
function countdown(n) {
  // 1. BASE CASE: Stop when n <= 0
  if (n <= 0) {
    console.log("Liftoff! [LAUNCH]");
    return;
  }

  console.log(n);

  // 2. RECURSIVE STEP: Call self with smaller sub-problem (n - 1)
  countdown(n - 1);
}

countdown(3); // Logs: 3, 2, 1, "Liftoff! [LAUNCH]"
```

Code Listing 19.1: A Simple Countdown Using Recursion

19.2 Tracing the Call Stack

Each recursive call creates a new **execution context** (stack frame) on the Call Stack. The frame stores the function's local variables and its return address:

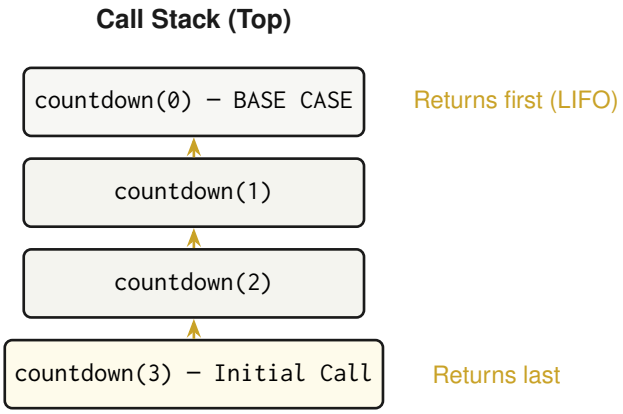


Figure 19.1: Call Stack Frames During Recursive Countdown

19.3 Classic Recursive Algorithms

```
// Factorial: n! = n * (n-1) * ... * 1
function factorial(n) {
  if (n <= 1) return 1;        // Base case
  return n * factorial(n - 1); // Recursive step
}

console.log(factorial(5));  // 120 (5 * 4 * 3 * 2 * 1)
console.log(factorial(10)); // 3628800

// Power: base^exponent using recursion
function power(base, exponent) {
  if (exponent === 0) return 1; // Any number to the 0th power is 1
  return base * power(base, exponent - 1);
}

console.log(power(2, 10)); // 1024
```

Code Listing 19.2: Factorial and Power Functions

```
// Sum of digits: 1234 -> 1 + 2 + 3 + 4 = 10
function sumOfDigits(n) {
  n = Math.abs(n); // Handle negative numbers
  if (n < 10) return n; // Base case: single digit
  return (n % 10) + sumOfDigits(Math.floor(n / 10));
}

console.log(sumOfDigits(1234)); // 10
console.log(sumOfDigits(9999)); // 36
```

```

// Recursive palindrome check
function isPalindrome(str) {
  if (str.length <= 1) return true; // Base case
  if (str[0] !== str[str.length - 1]) return false;
  return isPalindrome(str.slice(1, -1)); // Check inner substring
}

console.log(isPalindrome("racecar")); // true
console.log(isPalindrome("fedi"));    // false
```

Code Listing 19.3: Sum of Digits and Palindrome Check

19.4 Processing Hierarchical Data

Recursion shines when processing **nested, hierarchical** data structures where the depth is unknown at compile time:

```
// Count all keys in a deeply nested object (any depth)
function countAllKeys(obj) {
  let count = 0;
  for (const key in obj) {
    count++; // Count this key
    if (typeof obj[key] === "object" && obj[key] !== null) {
      count += countAllKeys(obj[key]); // Recurse into nested object
    }
  }
  return count;
}

const fediConfig = {
  app: {
    name: "FediApp",
    version: "2.0",
    features: {
      auth: { provider: "OAuth", mfa: true },
      analytics: { enabled: false }
    }
  },
  database: { host: "localhost", port: 5432 }
};

console.log(countAllKeys(fediConfig)); // 12 total keys at all levels
```

Code Listing 19.4: Deep Object Key Counter

```
// Flatten a deeply nested array to any depth
function deepFlatten(arr) {
  const result = [];
  for (const item of arr) {
    if (Array.isArray(item)) {
      result.push(...deepFlatten(item)); // Recurse and spread
    } else {
```

```
      result.push(item);
    }
  }
  return result;
}

const nested = [1, [2, [3, [4, [5]]]], 6, [[7]]];
console.log(deepFlatten(nested)); // [1, 2, 3, 4, 5, 6, 7]
```

Code Listing 19.5: Recursive Array Flattening

19.5 Stack Overflow and Maximum Call Stack Size

The JavaScript Call Stack has a finite size (typically 10,000–25,000 frames depending on the engine). If a recursive function calls itself too many times without reaching a base case (or if the input is simply too large for the available stack depth), the engine throws a `RangeError: Maximum call stack size exceeded`:

```
// DANGER: Missing base case causes infinite recursion!
function infiniteLoop() {
  return infiniteLoop(); // Never stops!
}
// infiniteLoop(); // RangeError: Maximum call stack size exceeded

// DANGER: Input too large for recursive depth
// factorial(100000); // Stack overflow even with correct base case

// SOLUTION: Convert to iteration for very large inputs
function factorialIterative(n) {
  let result = 1;
  for (let i = 2; i <= n; i++) {
    result *= i;
  }
  return result;
}
```

Code Listing 19.6: Causing and Preventing Stack Overflow

19.6 Tail Call Optimization (TCO)

A **tail call** occurs when a function's very last action is to call another function (or itself). In theory, the engine can reuse the current stack frame instead of creating a new one, enabling infinite recursion depth with $O(1)$ stack space:

```
// NOT tail-recursive: multiplication happens AFTER the recursive call returns
function factorialBad(n) {
  if (n <= 1) return 1;
  return n * factorialBad(n - 1); // Must multiply after call returns
}

```

```
// Tail-recursive: recursive call is the LAST operation (result accumulated in parameter)
function factorialTail(n, accumulator = 1) {
  if (n <= 1) return accumulator;
  return factorialTail(n - 1, n * accumulator); // Nothing to do after this call
}

console.log(factorialTail(5)); // 120
```

Code Listing 19.7: Tail-Recursive Factorial

COMMON MISTAKE

As of 2026, only Safari's JavaScriptCore engine implements proper tail call optimization (ES2015 spec). V8 (Chrome/Node.js) and SpiderMonkey (Firefox) do **not** implement TCO. For production code that processes very large inputs, convert tail-recursive functions to iterative loops manually.

19.7 Recursion vs Iteration

Table 19.1: Recursion vs Iteration Tradeoffs

Criterion	Recursion	Iteration
Readability	Often cleaner for trees/graphs	Cleaner for linear sequences
Stack Usage	$O(n)$ stack frames	$O(1)$ stack space
Stack Overflow Risk	Yes, for deep recursion	No
Performance	Function call overhead	Generally faster
Natural Fit	Trees, graphs, nested data	Arrays, counters, ranges

MINI CHALLENGE

The Directory Tree Printer. Write a recursive function `printTree(node, indent = "")` that accepts a file system tree object like `\{ name: "src", children: [\{ name: "index.js"\}, \{ name: "utils", children: [...] \}] \}` and prints an indented tree with `"---␣"` prefixes showing the hierarchy, similar to the Unix `tree` command output.

19.8 Exercises

E1. [Level 1] Write a recursive function `sumRange(n)` returning $1 + 2 + \cdots + n$.

E2. [Level 1] Write a recursive function `reverseString(str)` without using array methods.

E3. [Level 2] Write a recursive `deepClone(obj)` that handles nested objects and arrays.

E4. [Level 2] Implement a recursive binary search: `binarySearch(sortedArr, target, lo, hi)`.

E5. [Level 3] Write a recursive function `permutations(str)` returning all permutations of a string's characters.

E6. [Level 4] Implement a recursive Merge Sort: split the array in half, recursively sort each half, then merge the sorted halves.

19.9 Quiz

Q1. What are the two required components of every recursive function?
Q2. What error is thrown when recursion exceeds the Call Stack limit?
Q3. Why is recursion the natural approach for processing tree data structures?
Q4. What is a tail call, and which JavaScript engine currently supports TCO?
Q5. How can you convert a recursive function to iteration for safety?

CHAPTER SUMMARY

- Every recursive function needs a base case (stop condition) and a recursive step (self-call with smaller input).
- Each recursive call adds a stack frame; exceeding the limit throws `RangeError`.
- Recursion is the natural fit for trees, graphs, nested objects, and divide-and-conquer algorithms.
- For very deep recursion, convert to iteration or use tail-recursive patterns.

INTERVIEW CORNER

Q: When should you use recursion vs iteration? — Use recursion when the problem has a naturally recursive structure (tree traversal, nested data, backtracking, divide-and-conquer). Use iteration when processing linear sequences, when stack depth is a concern, or when performance is critical. Many recursive solutions can be converted to iterative ones using an explicit stack.

Q: How do you debug a recursive function? — Add a `depth` parameter and log it at each call to trace the recursion depth. Use `console.group()` and `console.groupEnd()` to visualize the nesting. Set a conditional breakpoint in DevTools that triggers only at specific recursion depths.

20 MODULE

Data Structures

A **data structure** is a specialized format for organizing, storing, and retrieving data efficiently in computer memory. While JavaScript provides built-in Arrays and Objects, understanding classical data structures — **Stacks, Queues, Linked Lists, Trees, and Hash Tables** — is what separates script writers from software engineers. In this module, you will implement each data structure from first principles using modern JavaScript classes, examine hash collisions, and analyze their performance tradeoffs.

LEARNING OBJECTIVES

- Understand the difference between contiguous memory and linked pointer structures
- Implement a Last-In, First-Out (LIFO) **Stack** for parsing and history tracking
- Implement a First-In, First-Out (FIFO) **Queue** with $O(1)$ operations
- Build Singly and Doubly **Linked Lists** and manage node pointers
- Construct a **Binary Search Tree (BST)** and maintain the BST ordering invariant
- Understand **Hash Tables**, design hash functions, and resolve collisions using Separate Chaining
- Select the optimal data structure based on time and space complexity classes

20.1 The Stack (LIFO)

A **Stack** is a Last-In, First-Out (LIFO) data structure. Think of a stack of dinner plates: you add new plates to the top, and you remove plates from the top.

```
class Stack {
  #items = [];

  push(element) {
    this.#items.push(element);
  }

  pop() {
    if (this.isEmpty()) return null;
    return this.#items.pop();
  }

  peek() {
    if (this.isEmpty()) return null;
    return this.#items[this.#items.length - 1];
  }

```

```
  isEmpty() {
    return this.#items.length === 0;
  }

  get size() {
    return this.#items.length;
  }
}

const undoStack = new Stack();
undoStack.push("Type 'Hello'");
undoStack.push("Type ' World'");
console.log(undoStack.pop()); // "Type ' World'" (Last action undone first)
```

Code Listing 20.1: Implementing a Stack with modern JS Classes

20.2 The Queue (FIFO)

A **Queue** is a First-In, First-Out (FIFO) data structure. Like a checkout line at a store, the first person to arrive is the first person served.

```
class Queue {
  #items = {};
  #headIndex = 0;
  #tailIndex = 0;

  // Enqueue: Add item to the back O(1)
  enqueue(item) {
    this.#items[this.#tailIndex] = item;
    this.#tailIndex++;
  }

  // Dequeue: Remove item from the front O(1)
  dequeue() {
    if (this.isEmpty()) return null;
    const item = this.#items[this.#headIndex];
    delete this.#items[this.#headIndex];
    this.#headIndex++;
    return item;
  }

  peek() {
    return this.isEmpty() ? null : this.#items[this.#headIndex];
  }

  isEmpty() {
    return this.#tailIndex - this.#headIndex === 0;
  }

  get size() {
    return this.#tailIndex - this.#headIndex;
  }
```

```
}
```

Code Listing 20.2: Implementing an O(1) Queue

20.3 Linked Lists: Singly and Doubly

Unlike arrays which store elements contiguously in memory, a **Linked List** consists of independent **Node** objects where each node holds a `value` and pointers referencing other nodes in the chain.

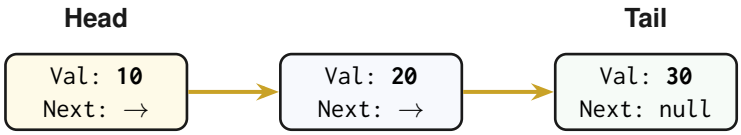


Figure 20.1: Singly Linked List Chain Architecture

```
class DoublyListNode {
  constructor(value) {
    this.value = value;
    this.next = null;
    this.prev = null;
  }
}

class DoublyLinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
    this.size = 0;
  }

  append(value) {
    const newNode = new DoublyListNode(value);
    if (!this.head) {
      this.head = newNode;
      this.tail = newNode;
    } else {
      this.tail.next = newNode;
      newNode.prev = this.tail;
      this.tail = newNode;
    }
    this.size++;
  }

  removeNode(node) {
    if (!node) return;
    if (node.prev) node.prev.next = node.next;
    else this.head = node.next; // Node was head

    if (node.next) node.next.prev = node.prev;
    else this.tail = node.prev; // Node was tail

    this.size--;
```

```
  }
}
```

Code Listing 20.3: Implementing a Doubly Linked List Node and List

20.4 Binary Search Trees (BST)

A **Binary Search Tree** is a hierarchical tree structure where each node has at most two children, satisfying the **BST Invariant**: all values in the left subtree are less than the parent, and all values in the right subtree are greater than the parent.

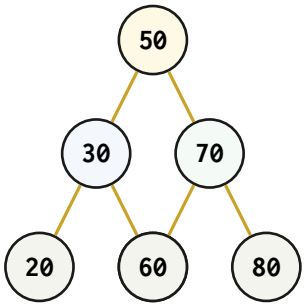


Figure 20.2: A Balanced Binary Search Tree (BST)

```
class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
}

class BinarySearchTree {
  constructor() {
    this.root = null;
  }

  insert(value) {
    const newNode = new TreeNode(value);
    if (!this.root) {
      this.root = newNode;
      return this;
    }

    let current = this.root;
    while (true) {
      if (value === current.value) return undefined; // No duplicates
      if (value < current.value) {
        if (!current.left) {
          current.left = newNode;
          return this;
        }
        current = current.left;
```

```
      } else {
        if (!current.right) {
          current.right = newNode;
          return this;
        }
        current = current.right;
      }
    }
  }

  contains(value) {
    let current = this.root;
    while (current) {
      if (value === current.value) return true;
      current = value < current.value ? current.left : current.right;
    }
    return false;
  }
}
```

Code Listing 20.4: Binary Search Tree Implementation

20.5 Hash Tables and Collision Resolution

A **Hash Table** (or Hash Map) stores key-value pairs. It uses a **Hash Function** to convert keys into integer array indices, allowing $O(1)$ average-case insertion and lookup times. When two different keys hash to the same index, a **Hash Collision** occurs. The most common collision resolution strategy is **Separate Chaining**, where each bucket contains a linked list of entries sharing that index.

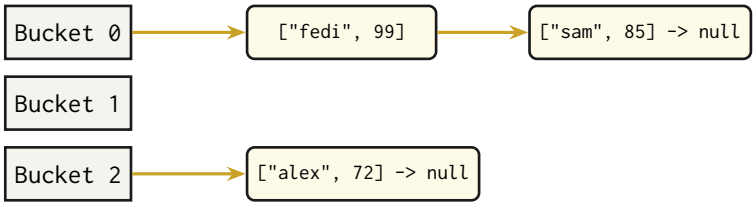


Figure 20.3: Hash Table with Separate Chaining Collision Resolution

```
class HashTable {
  constructor(size = 50) {
    this.buckets = new Array(size).fill(null).map(() => []);
    this.size = size;
  }

  // Polynomial rolling hash function
  _hash(key) {
    let total = 0;
    const WEIRD_PRIME = 31;
    for (let i = 0; i < Math.min(key.length, 100); i++) {
      const char = key.charCodeAt(i);
      total = (total * WEIRD_PRIME + char) % this.size;
```

```
    }
    return total;
  }

  set(key, value) {
    const index = this._hash(key);
    const bucket = this.buckets[index];
    const foundEntry = bucket.find(entry => entry[0] === key);

    if (foundEntry) {
      foundEntry[1] = value; // Update existing key
    } else {
      bucket.push([key, value]); // Insert new key-value pair
    }
  }

  get(key) {
    const index = this._hash(key);
    const bucket = this.buckets[index];
    const foundEntry = bucket.find(entry => entry[0] === key);
    return foundEntry ? foundEntry[1] : undefined;
  }
}

const scores = new HashTable(10);
scores.set("fedi", 99);
scores.set("sam", 85);
console.log(scores.get("fedi")); // 99
```

Code Listing 20.5: Hash Table with Separate Chaining

MINI CHALLENGE

The Balanced Parentheses Validator. Using a `Stack`, write a function `isValidParentheses(str)` that returns `true` if brackets `()`, `{}`, and `[]` are properly opened and closed in correct order (e.g. `"{[()]}"` is valid, but `"{[(])}"` is invalid).

20.6 Exercises

E1. [Level 1] Implement a `min()` method on a Stack that retrieves the minimum element in $O(1)$ time.

E2. [Level 1] Implement a `size` getter and `toArray()` converter for a Singly Linked List.

E3. [Level 2] Reverse a Singly Linked List in place by manipulating pointers in $O(n)$ time and $O(1)$ space.

E4. [Level 2] Implement an In-Order traversal method on a Binary Search Tree that returns all values in sorted ascending order.

E5. [Level 3] Implement a Doubly Linked List with `prev` and `next` pointers and $O(1)$ deletion of any given node.

E6. [Level 4] Build a custom Hash Table class with dynamic resizing (rehashing when load factor > 0.75) and separate chaining for collision resolution.

20.7 Quiz

Q1. What is the difference between LIFO and FIFO? Which data structures model each?
Q2. Why does `Array.prototype.shift()` take $O(n)$ time, while an object-based Queue `dequeue()` takes $O(1)$?
Q3. What is the time complexity of searching a value in a balanced Binary Search Tree vs an unindexed array?
Q4. What is a hash collision in a Hash Table and how does separate chaining resolve it?
Q5. What happens to a Binary Search Tree's performance if elements are inserted in strictly sorted order?

CHAPTER SUMMARY

- Stacks (LIFO) model undo histories, Call Stacks, and syntactic parsing.
- Queues (FIFO) model task scheduling, background workers, and event queues.
- Linked Lists allow efficient $O(1)$ insertions and deletions without array resizing.
- Binary Search Trees enable fast $O(\log n)$ search, insertion, and sorted traversals.

INTERVIEW CORNER

Q: How do you detect a cycle in a Linked List? — Use **Floyd's Cycle-Finding Algorithm (Tortoise and Hare)**: maintain two pointers, slow (advances 1 node) and fast (advances 2 nodes). If a cycle exists, the fast pointer will eventually lap and equal the slow pointer ($O(n)$ time, $O(1)$ space).

Q: What is the load factor of a Hash Table? — The load factor (α) is the ratio of the number of occupied slots (n) to the total number of buckets (m): $\alpha = n/m$. When the load factor exceeds a threshold (typically 0.75), the table's average search time degrades due to increased collisions, triggering a resizing and rehashing phase to maintain $O(1)$ complexity.

21 MODULE

Algorithms and Big O Notation

An **algorithm** is a step-by-step procedure for solving a computational problem. Two algorithms can produce the exact same correct answer while differing drastically in speed and memory efficiency: one might finish in 2 milliseconds, while the other takes 4 hours on large datasets. In this module, you will learn to measure algorithmic efficiency using **Big O Notation**, compare sorting algorithms, and master essential computational patterns: Two Pointers, Fast/Slow Pointers, and Sliding Windows.

LEARNING OBJECTIVES

- Understand Big O Notation and analyze Time and Space complexity
- Differentiate complexity classes: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$
- Implement **Linear Search** ($O(n)$) and logarithmic **Binary Search** ($O(\log n)$)
- Understand and code sorting algorithms: Bubble, Selection, Merge, and Quick Sort
- Compare sorting algorithms across best, average, and worst-case scenarios
- Master common algorithmic design patterns: **Two Pointers** (Opposite & Fast/Slow) and **Sliding Window** (Fixed & Dynamic)
- Solve classic interview problems with optimal space-time tradeoffs

21.1 Understanding Big O Notation

Big O Notation describes how an algorithm's execution time or memory usage scales as the input size n approaches infinity. It characterizes the **worst-case upper bound**.

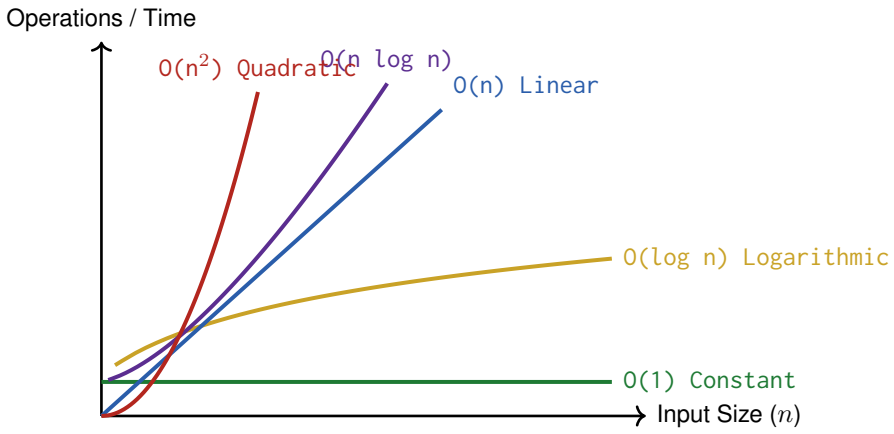


Figure 21.1: Comparison of Common Big O Complexity Curves

21.2 Searching Algorithms

Table 21.1: Common Complexity Classes and Examples

Complexity	Name	Real-World Example
$O(1)$	Constant	Array index lookup (`arr[0]`), Map `get()`
$O(\log n)$	Logarithmic	Binary search in sorted array, BST lookup
$O(n)$	Linear	Linear scan, `Array.prototype.find()`, `map()`
$O(n \log n)$	Linearithmic	Merge Sort, Quick Sort, native `Array.prototype.sort()`
$O(n^2)$	Quadratic	Nested `for` loops, Bubble Sort
$O(2^n)$	Exponential	Naive recursive Fibonacci

```
// 1. Linear Search: O(n) Time, works on unsorted arrays
function linearSearch(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) return i;
  }
  return -1;
}

// 2. Binary Search: O(log n) Time, REQUIRES SORTED ARRAY!
function binarySearch(sortedArr, target) {
  let left = 0;
  let right = sortedArr.length - 1;

  while (left <= right) {
    const mid = Math.floor((left + right) / 2);
    if (sortedArr[mid] === target) {
      return mid; // Found target at index mid!
    }
    if (sortedArr[mid] < target) {
      left = mid + 1;  // Search right half
    } else {
      right = mid - 1; // Search left half
    }
  }
  return -1; // Target not found
}

const numbers = [12, 24, 35, 47, 59, 68, 77, 89, 94];
console.log(binarySearch(numbers, 68)); // 5 (Found in only 3 comparisons!)
```

Code Listing 21.1: Linear Search vs Binary Search

21.3 Sorting Algorithms

Sorting is the process of arranging data in a specific order. While simple sorting algorithms like Bubble Sort or Selection Sort run in $O(n^2)$ time, divide-and-conquer sorting algorithms such as Merge Sort and Quick Sort achieve $O(n \log n)$ average complexity:

```
function mergeSort(arr) {
```

Table 21.2: Comparison of Sorting Algorithms

Algorithm	Best Time	Average Time	Worst Time	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$ (temp arrays)
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ (recursion)

```
  if (arr.length <= 1) return arr; // Base case

  const mid = Math.floor(arr.length / 2);
  const left = mergeSort(arr.slice(0, mid));
  const right = mergeSort(arr.slice(mid));

  return merge(left, right);
}

function merge(left, right) {
  const sorted = [];
  let l = 0, r = 0;

  while (l < left.length && r < right.length) {
    if (left[l] <= right[r]) {
      sorted.push(left[l]);
      l++;
    } else {
      sorted.push(right[r]);
      r++;
    }
  }
  return sorted.concat(left.slice(l)).concat(right.slice(r));
}
```

Code Listing 21.2: Merge Sort Implementation: O(n log n)

Quick Sort Partitioning (Pivot = 5):
Values $<$ 5 move to the left side.
Values $>$ 5 move to the right side.

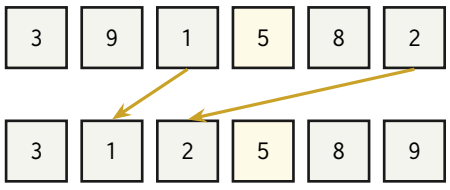


Figure 21.2: Quick Sort Partitioning Concept

21.4 The Two-Pointer Pattern: Opposite and Fast/Slow

The two-pointer technique uses two moving index variables to scan data collections efficiently. It has two primary variations:

1. **Opposite End Pointers**: Pointers start at indices `0` and `length - 1` and move toward each other. Excellent for checking palindromes or finding target pairs in sorted lists.
2. **Fast and Slow Pointers**: Pointers move at different speeds (e.g. slow moves 1 step, fast moves 2). Known as the Tortoise and Hare algorithm, it is ideal for detecting cycles in linked lists or finding node midpoints.

```
function hasPairWithSumSorted(sortedArr, target) {
  let left = 0;
  let right = sortedArr.length - 1;

  while (left < right) {
    const currentSum = sortedArr[left] + sortedArr[right];
    if (currentSum === target) return true;
    if (currentSum < target) {
      left++;  // Need a larger sum -> move left pointer right
    } else {
      right--; // Need a smaller sum -> move right pointer left
    }
  }
  return false;
}
```

Code Listing 21.3: Two Pointers: Target Pair in Sorted Array

21.5 The Sliding Window Pattern: Fixed and Dynamic

The sliding window pattern converts nested loop algorithms over contiguous subarrays into single-pass algorithms. It also has two variations:

1. **Fixed-Size Window**: The window maintains a constant width k as it slides from left to right.
2. **Dynamic-Size Window**: The window grows and shrinks dynamically based on constraints (e.g., finding the shortest subarray that sums to at least S).

```
function longestUniqueSubstringLength(str) {
  let maxLength = 0;
  let left = 0;
  const seenChars = new Set();

  for (let right = 0; right < str.length; right++) {
    // Shrink window from the left until the duplicate char is removed
    while (seenChars.has(str[right])) {
      seenChars.delete(str[left]);
      left++;
    }
    seenChars.add(str[right]);
    maxLength = Math.max(maxLength, right - left + 1);
  }

  return maxLength;
}

console.log(longestUniqueSubstringLength("abcabcbb")); // 3 ("abc")
```

```
20 console.log(longestUniqueSubstringLength("bbbbb"));     // 1 ("b")
```

Code Listing 21.4: Dynamic Sliding Window: Longest Substring without Repeating Characters

MINI CHALLENGE

The Classic Two Sum Problem ($O(n)$ Time). Given an unsorted array of numbers and a target sum, find the indices of the two numbers that add up to the target in $O(n)$ linear time using a `Map` hash table cache.

21.6 Exercises

E1. [Level 1] Determine the Big O time complexity of: (a) iterating an array with `forEach`, (b) `array.unshift()`, (c) checking `map.has()`.
E2. [Level 1] Implement Bubble Sort and count the number of swaps made during execution.
E3. [Level 2] Implement Quick Sort using a pivot partition helper function.
E4. [Level 2] Write a function `maxSubarraySum(arr, k)` that finds the maximum sum of any contiguous subarray of size k using the **Sliding Window** technique ($O(n)$).
E5. [Level 3] Solve the **3Sum** problem ($a + b + c = 0$) in $O(n^2)$ time using sorting and two pointers.
E6. [Level 4] Implement an algorithm to find the k-th largest element in an unsorted array in average $O(n)$ time using **QuickSelect**.

21.7 Quiz

Q1. What is the Big O time complexity of Binary Search, and what precondition must the array meet?
Q2. Why is an $O(n \log n)$ algorithm vastly superior to an $O(n^2)$ algorithm for $n = 1,000,000$?
Q3. What is the difference between Time Complexity and Space Complexity?
Q4. How does the Sliding Window pattern optimize contiguous subarray calculations from $O(n \cdot k)$ to $O(n)$?
Q5. What is the worst-case time complexity of Quick Sort, and when does it occur?

CHAPTER SUMMARY

- Big O Notation provides a hardware-independent metric for algorithmic scalability.
- Binary Search delivers $O(\log n)$ logarithmic lookups on sorted collections.
- Merge Sort and Quick Sort achieve $O(n \log n)$ sorting via divide-and-conquer.
- Master patterns like Two-Pointers and Sliding Windows to reduce $O(n^2)$ bottlenecks to $O(n)$.

INTERVIEW CORNER

Q: What is the Two Sum problem and how do you solve it in O(n) time? — Maintain a Hash Map of `{ complement: index }`. As you iterate through the array, check if `map.has(target - currentNum)`. If present, you found the pair in a single pass ($O(n)$ time, $O(n)$ space).

Q: What is the difference between Stable and Unstable sorting? — A sorting algorithm

is **stable** if it preserves the relative order of elements with equal values. For example, if you sort a list of transactions by amount, a stable sort keeps transactions with identical amounts ordered by their original timestamps, while an unstable sort might reorder them.

PART IV

Tier 4 — Master

TIER 4 — MASTER

The final ascent: graphs and trees, dynamic programming, functional programming, and the asynchronous heart of JavaScript — the event loop, promises, and async/await. Master these and you can hold your own in any technical interview.

22 MODULE

Graphs and Trees

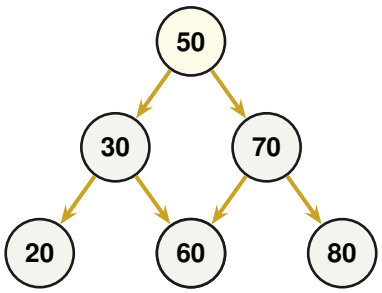
Arrays and Linked Lists are **linear** data structures: each element has at most one predecessor and one successor. Real-world relationships, however, are rarely linear. Social networks, file systems, organizational hierarchies, road maps, and dependency graphs all model entities connected in complex, non-linear topologies. In this module, you will master the two foundational non-linear data structures: **Trees** and **Graphs**, and implement their traversal algorithms from scratch in JavaScript.

LEARNING OBJECTIVES

- Understand tree terminology: root, parent, child, leaf, depth, height, subtree
- Implement Binary Trees with a `TreeNode` class and recursive traversals
- Build a Binary Search Tree (BST) with `insert`, `search`, `findMin`, `findMax`
- Analyze BST operations: $O(\log n)$ average-case vs $O(n)$ worst-case (degenerate tree)
- Understand graph terminology: vertex, edge, directed, undirected, weighted, cyclic
- Represent graphs using Adjacency Lists and Adjacency Matrices
- Implement Breadth-First Search (BFS) with a Queue for shortest-path discovery
- Implement Depth-First Search (DFS) with a Stack or recursion for deep exploration

22.1 Tree Fundamentals and Terminology

A **tree** is a hierarchical data structure consisting of nodes connected by edges. Every tree has exactly one **root node** with zero or more child subtrees.



Root: 50 (depth 0)
Internal Nodes: 30, 70
Leaf Nodes: 20, 40, 60, 80
Height: 2 levels
Type: Binary Search Tree

Figure 22.1: A Balanced Binary Search Tree with Labeled Terminology

Key terminology:

Root The topmost node with no parent (depth 0).
Parent / Child A node directly above / below another node via an edge.
Leaf A node with no children (an endpoint of the tree).
Depth The number of edges from the root to a given node.

Height The number of edges on the longest path from the root to a leaf.
Subtree Any node and all of its descendants form a valid subtree.

22.2 Binary Tree Node Implementation

```
class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;   // Left child subtree
    this.right = null;  // Right child subtree
  }
}

// Manually constructing a small tree:
const root = new TreeNode(50);
root.left = new TreeNode(30);
root.right = new TreeNode(70);
root.left.left = new TreeNode(20);
root.left.right = new TreeNode(40);
root.right.left = new TreeNode(60);
root.right.right = new TreeNode(80);
```

Code Listing 22.1: Binary Tree Node Class

22.3 Tree Traversal Algorithms

There are three classic recursive traversal orders for binary trees:

Table 22.1: Binary Tree Traversal Orders

Traversal	Visit Order	Output (above BST)	Use Case
Inorder	Left, Root, Right	20, 30, 40, 50, 60, 70, 80	Sorted output from BST
Preorder	Root, Left, Right	50, 30, 20, 40, 70, 60, 80	Tree cloning / serialization
Postorder	Left, Right, Root	20, 40, 30, 60, 80, 70, 50	Safe deletion (children first)

```
function inorderTraversal(node, result = []) {
  if (node === null) return result;
  inorderTraversal(node.left, result);   // 1. Visit left subtree
  result.push(node.value);              // 2. Visit current node
  inorderTraversal(node.right, result); // 3. Visit right subtree
  return result;
}

function preorderTraversal(node, result = []) {
  if (node === null) return result;
  result.push(node.value);              // 1. Visit current node
  preorderTraversal(node.left, result); // 2. Visit left subtree
  preorderTraversal(node.right, result);// 3. Visit right subtree
  return result;
}
```

```

function postorderTraversal(node, result = []) {
  if (node === null) return result;
  postorderTraversal(node.left, result); // 1. Visit left subtree
  postorderTraversal(node.right, result);// 2. Visit right subtree
  result.push(node.value);              // 3. Visit current node
  return result;
}

console.log(inorderTraversal(root));  // [20, 30, 40, 50, 60, 70, 80]
console.log(preorderTraversal(root)); // [50, 30, 20, 40, 70, 60, 80]
```

Code Listing 22.2: Recursive Tree Traversal Implementations

22.4 Binary Search Tree Operations

A **Binary Search Tree (BST)** enforces a strict ordering invariant: for every node N, all values in the left subtree are strictly less than N, and all values in the right subtree are strictly greater than N.

```
class BinarySearchTree {
  constructor() {
    this.root = null;
  }

  insert(value) {
    const newNode = new TreeNode(value);
    if (!this.root) {
      this.root = newNode;
      return;
    }
    let current = this.root;
    while (true) {
      if (value < current.value) {
        if (!current.left) { current.left = newNode; return; }
        current = current.left;
      } else {
        if (!current.right) { current.right = newNode; return; }
        current = current.right;
      }
    }
  }

  search(value) {
    let current = this.root;
    while (current) {
      if (value === current.value) return current;
      current = value < current.value ? current.left : current.right;
    }
    return null; // Not found
  }

  findMin(node = this.root) {
    while (node && node.left) {
```

```
      node = node.left;
    }
    return node ? node.value : null;
  }

  findMax(node = this.root) {
    while (node && node.right) {
      node = node.right;
    }
    return node ? node.value : null;
  }
}

const bst = new BinarySearchTree();
[50, 30, 70, 20, 40, 60, 80].forEach(v => bst.insert(v));
console.log(bst.search(40));   // TreeNode { value: 40, ... }
console.log(bst.findMin());    // 20
console.log(bst.findMax());    // 80
```

Code Listing 22.3: BST Insert, Search, and FindMin

COMMON MISTAKE

A BST degenerates into a linked list when values are inserted in sorted order (e.g., 1, 2, 3, 4, 5), causing all operations to degrade from $O(\log n)$ to $O(n)$. Self-balancing trees like AVL Trees and Red-Black Trees prevent this by automatically rebalancing after each insertion.

22.5 Graph Fundamentals and Terminology

A **graph** $G = (V, E)$ consists of a set of **vertices** (nodes) V and a set of **edges** (connections) E. Unlike trees, graphs may contain cycles, multiple paths between nodes, and disconnected components.

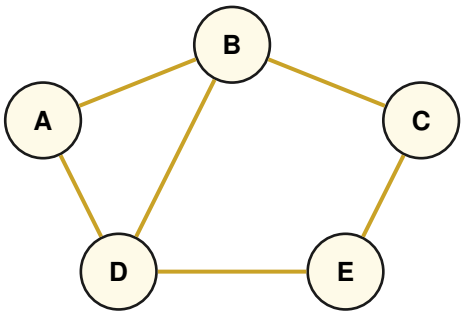


Figure 22.2: An Undirected Graph with 5 Vertices and 6 Edges

22.6 Graph Representation: Adjacency List

```
class Graph {
  constructor() {
    this.adjacencyList = new Map();
```

Table 22.2: Adjacency Matrix vs Adjacency List

Criterion	Adjacency Matrix	Adjacency List
Space Complexity	$O(V^2)$	$O(V + E)$
Edge Lookup	$O(1)$	$O(\text{degree})$
Add Vertex	$O(V^2)$ rebuild	$O(1)$
Best For	Dense graphs	Sparse graphs (most real-world)

```
  }

  addVertex(vertex) {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, new Set());
    }
  }

  addEdge(v1, v2) {
    this.addVertex(v1);
    this.addVertex(v2);
    this.adjacencyList.get(v1).add(v2);
    this.adjacencyList.get(v2).add(v1); // Remove for directed graph
  }

  getNeighbors(vertex) {
    return this.adjacencyList.get(vertex) || new Set();
  }
}

const socialNetwork = new Graph();
socialNetwork.addEdge("Fedi", "Sam");
socialNetwork.addEdge("Fedi", "Alex");
socialNetwork.addEdge("Sam", "Jordan");
socialNetwork.addEdge("Alex", "Jordan");

console.log(socialNetwork.getNeighbors("Fedi")); // Set {"Sam", "Alex"}
```

Code Listing 22.4: Graph Class with Adjacency List

22.7 Breadth-First Search (BFS)

BFS explores a graph **level by level** outward from a starting vertex, using a **FIFO Queue**. It guarantees discovering the **shortest path** (fewest edges) in unweighted graphs.

```
function bfs(graph, startVertex) {
  const visited = new Set([startVertex]);
  const queue = [startVertex];
  const traversalOrder = [];
  const parentMap = new Map(); // For path reconstruction!

  while (queue.length > 0) {
    const current = queue.shift(); // Dequeue from FRONT (FIFO)
    traversalOrder.push(current);

```

```
    for (const neighbor of graph.getNeighbors(current)) {
      if (!visited.has(neighbor)) {
        visited.add(neighbor);
        parentMap.set(neighbor, current);
        queue.push(neighbor); // Enqueue at BACK
      }
    }
  }

  return { traversalOrder, parentMap };
}

const { traversalOrder } = bfs(socialNetwork, "Fedi");
console.log(traversalOrder); // ["Fedi", "Sam", "Alex", "Jordan"]
```

Code Listing 22.5: BFS Implementation with Shortest Path Tracking

22.8 Depth-First Search (DFS)

DFS explores a graph by diving as **deep as possible** into one branch before backtracking, using a **LIFO Stack** (or recursion, which uses the Call Stack implicitly).

```
// Iterative DFS using explicit Stack
function dfsIterative(graph, startVertex) {
  const visited = new Set();
  const stack = [startVertex];
  const traversalOrder = [];

  while (stack.length > 0) {
    const current = stack.pop(); // Pop from TOP (LIFO)
    if (visited.has(current)) continue;
    visited.add(current);
    traversalOrder.push(current);

    for (const neighbor of graph.getNeighbors(current)) {
      if (!visited.has(neighbor)) {
        stack.push(neighbor);
      }
    }
  }
  return traversalOrder;
}

// Recursive DFS
function dfsRecursive(graph, vertex, visited = new Set()) {
  visited.add(vertex);
  console.log(`Visiting: ${vertex}`);

  for (const neighbor of graph.getNeighbors(vertex)) {
    if (!visited.has(neighbor)) {
      dfsRecursive(graph, neighbor, visited);
    }
  }
```

```
}
```

Code Listing 22.6: DFS: Iterative and Recursive Implementations

Table 22.3: BFS vs DFS Comparison

Algorithm	Data Structure	Traversal Order	Primary Use Cases
BFS	FIFO Queue	Level by level	Shortest path, peer discovery, web crawling
DFS	LIFO Stack / Recursion	Deep exploration	Maze solving, cycle detection, topological sort

NOTE

Both BFS and DFS have time complexity $O(V + E)$ where V is the number of vertices and E is the number of edges. The key difference is the **order** in which they discover nodes, which determines which algorithm is optimal for a given problem.

MINI CHALLENGE

The Social Connection Distance Finder. Using the social network graph above, implement a function `shortestPath(graph, start, target)` that uses BFS to find the minimum number of edges between two people (the "degrees of separation"). Reconstruct and return the actual path as an array of vertex names.

22.9 Exercises

E1. [Level 1] Build a BST from the values [15, 10, 20, 8, 12, 17, 25] and print the inorder traversal.
E2. [Level 1] Create a graph with 6 vertices and run BFS from vertex "A". Print the traversal order.
E3. [Level 2] Implement a function `treeHeight(root)` that computes the height of a binary tree recursively.
E4. [Level 2] Implement a level-order traversal (BFS on a tree) that returns an array of arrays, one per level.
E5. [Level 3] Implement a function `hasCycle(graph)` that detects if an undirected graph contains a cycle using DFS.
E6. [Level 4] Implement BST node deletion handling all three cases: leaf node, node with one child, and node with two children (replace with inorder successor).

22.10 Quiz

Q1. What ordering invariant does a Binary Search Tree enforce?
Q2. What is the difference between a tree and a graph?
Q3. Why does BFS guarantee the shortest path in an unweighted graph while DFS does not?
Q4. What is the time complexity of searching for a value in a balanced BST vs a degenerate BST?
Q5. When would you prefer an Adjacency Matrix over an Adjacency List?

CHAPTER SUMMARY

- Trees are hierarchical, acyclic structures with a single root node.
- BST operations run in $O(\log n)$ average-case but degrade to $O(n)$ when unbalanced.
- Graphs model arbitrary relationships; Adjacency Lists are optimal for sparse networks.
- BFS (Queue) explores level-by-level for shortest paths; DFS (Stack) dives deep for exhaustive exploration.

INTERVIEW CORNER

Q: When would you use BFS vs DFS? — Use BFS when you need the shortest path in an unweighted graph (e.g., social network degrees of separation, maze shortest route). Use DFS when you need to explore all possible paths exhaustively (e.g., maze solving, cycle detection, topological sorting of build dependencies).

Q: What is the time and space complexity of BFS? — Time: $O(V + E)$ since every vertex and edge is examined once. Space: $O(V)$ for the visited set and queue, which in the worst case holds all vertices at the widest level of the graph.

23 MODULE

Dynamic Programming

Dynamic Programming (DP) is a powerful algorithmic optimization technique that transforms exponential-time recursive algorithms into polynomial-time solutions. The core insight is simple: if a problem can be decomposed into **overlapping subproblems** that share the same **optimal substructure**, you can solve each subproblem once, store the result, and look it up when needed again — eliminating redundant computation entirely. DP is the single most important algorithm design paradigm tested in senior technical interviews.

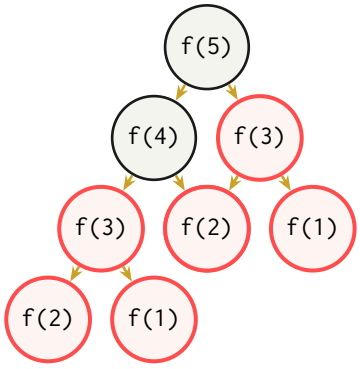
LEARNING OBJECTIVES

- Recognize the two prerequisites for DP: overlapping subproblems and optimal substructure
- Implement **Top-Down Memoization**: recursion with a lookup cache
- Implement **Bottom-Up Tabulation**: iterative table-filling from base cases
- Analyze time and space complexity improvements from exponential to polynomial
- Solve classic DP problems: Fibonacci, Climbing Stairs, Coin Change, Grid Paths, Longest Common Subsequence
- Optimize space complexity from $O(n)$ tables to $O(1)$ with rolling variables
- Visualize DP state transition diagrams and dependency graphs

23.1 When to Apply Dynamic Programming

A problem is a candidate for DP when it exhibits both:

1. **Overlapping Subproblems**: The recursive decomposition solves the same subproblems repeatedly. For example, naive Fibonacci computes `fib(3)` hundreds of times.
2. **Optimal Substructure**: The optimal solution to the full problem can be constructed from optimal solutions to its subproblems. For example, the shortest path from A to C via B is the shortest A-to-B path plus the shortest B-to-C path.



Red nodes are redundant overlapping subproblems.

f(3) computed 2 times
f(2) computed 3 times
f(1) computed 2 times

Total calls: $O(2^n)$
With DP: $O(n)$

Figure 23.1: Naive Fibonacci Recursion Tree Showing Overlapping Subproblems

23.2 Strategy 1: Top-Down Memoization

Memoization adds a cache (typically a `Map` or object) to a recursive function. Before computing a subproblem, check if its result already exists in the cache:

```
function fibMemo(n, memo = new Map()) {
  // 1. Check cache first
  if (memo.has(n)) return memo.get(n);

  // 2. Base cases
  if (n <= 0) return 0;
  if (n === 1) return 1;

  // 3. Recursive computation
  const result = fibMemo(n - 1, memo) + fibMemo(n - 2, memo);

  // 4. Store in cache BEFORE returning
  memo.set(n, result);
  return result;
}

console.log(fibMemo(50)); // 12586269025 (instant, not heat-death-of-universe!)
```

Code Listing 23.1: Fibonacci with Top-Down Memoization

NOTE

Memoization preserves the natural recursive structure of the problem, making it easier to write and reason about. The tradeoff is that it uses $O(n)$ Call Stack depth, which may cause `RangeError: Maximum call stack size exceeded` for very large inputs.

23.3 Strategy 2: Bottom-Up Tabulation

Tabulation eliminates recursion entirely. You build a table (array) starting from the base cases and iteratively fill it up to the target value:

```
function fibTab(n) {
  if (n <= 1) return n;

  const table = new Array(n + 1);
  table[0] = 0;
  table[1] = 1;

  for (let i = 2; i <= n; i++) {
    table[i] = table[i - 1] + table[i - 2];
  }

  return table[n];
}

```

```
console.log(fibTab(50)); // 12586269025
```

Code Listing 23.2: Fibonacci with Bottom-Up Tabulation

23.4 Space Optimization: Rolling Variables

When a DP recurrence only depends on the previous 1–2 values, you can reduce space from $O(n)$ to $O(1)$ using rolling variables:

```
function fibOptimized(n) {
  if (n <= 1) return n;
  let prev2 = 0;
  let prev1 = 1;

  for (let i = 2; i <= n; i++) {
    const current = prev1 + prev2;
    prev2 = prev1;
    prev1 = current;
  }

  return prev1;
}
```

Code Listing 23.3: Fibonacci with O(1) Space

23.5 Classic DP Problem: Climbing Stairs

Problem: You are climbing a staircase with n steps. Each time you can climb either 1 step or 2 steps. In how many distinct ways can you reach the top?

```
// 1. Memoized (Top-Down)
function climbStairsMemo(n, memo = new Map()) {
  if (n <= 1) return 1;
  if (memo.has(n)) return memo.get(n);
  const result = climbStairsMemo(n - 1, memo) + climbStairsMemo(n - 2, memo);
  memo.set(n, result);
  return result;
}

// 2. Tabulated (Bottom-Up)
function climbStairsTab(n) {
  if (n <= 1) return 1;
  const dp = [1, 1]; // dp[0] = 1 way, dp[1] = 1 way
  for (let i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[n];
}

// 3. Space-Optimized O(1)
```

```
function climbStairs(n) {
  if (n <= 1) return 1;
  let prev2 = 1, prev1 = 1;
  for (let i = 2; i <= n; i++) {
    const cur = prev1 + prev2;
    prev2 = prev1;
    prev1 = cur;
  }
  return prev1;
}

console.log(climbStairs(10)); // 89 distinct ways
```

Code Listing 23.4: Climbing Stairs: Three DP Approaches

23.6 Classic DP Problem: Coin Change

Problem: Given an array of coin denominations and a target amount, find the **minimum number of coins** needed to make exact change. Return -1 if impossible.

```
function coinChange(coins, amount) {
  // dp[i] = minimum coins needed to make amount i
  const dp = new Array(amount + 1).fill(Infinity);
  dp[0] = 0; // Base case: 0 coins needed for amount 0

  for (let i = 1; i <= amount; i++) {
    for (const coin of coins) {
      if (coin <= i) {
        dp[i] = Math.min(dp[i], dp[i - coin] + 1);
      }
    }
  }

  return dp[amount] === Infinity ? -1 : dp[amount];
}

console.log(coinChange([1, 5, 10, 25], 36)); // 3 (25 + 10 + 1)
console.log(coinChange([2], 3));             // -1 (impossible)
```

Code Listing 23.5: Coin Change with Bottom-Up Tabulation

23.7 Classic DP Problem: Unique Grid Paths

Problem: A robot starts at cell $(1, 1)$ of an $m \times n$ grid and can only move **right** or **down**. How many unique paths exist to reach cell (m, n)?

```
function uniquePaths(m, n) {
  // Create m x n grid initialized to 1
  const dp = Array.from({ length: m }, () => new Array(n).fill(1));

```

```
  // Fill grid: each cell = sum of cell above + cell to the left
  for (let r = 1; r < m; r++) {
    for (let c = 1; c < n; c++) {
      dp[r][c] = dp[r - 1][c] + dp[r][c - 1];
    }
  }

  return dp[m - 1][n - 1];
}

console.log(uniquePaths(3, 7)); // 28 unique paths
console.log(uniquePaths(3, 3)); // 6 unique paths
```

Code Listing 23.6: Unique Grid Paths with 2D Tabulation

23.8 Classic DP Problem: Longest Common Subsequence

Problem: Given two strings, find the length of their longest common subsequence (characters that appear in the same relative order, but not necessarily contiguously).

```
function longestCommonSubsequence(text1, text2) {
  const m = text1.length;
  const n = text2.length;
  // dp[i][j] = LCS length of text1[0..i-1] and text2[0..j-1]
  const dp = Array.from({ length: m + 1 }, () => new Array(n + 1).fill(0));

  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (text1[i - 1] === text2[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1] + 1; // Characters match: extend LCS
      } else {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]); // Take best from either
    ↪ direction
      }
    }
  }

  return dp[m][n];
}

console.log(longestCommonSubsequence("FEDIAPP", "FEDIDEP")); // 5 ("FEDIP")
```

Code Listing 23.7: Longest Common Subsequence (LCS)

MINI CHALLENGE

The Minimum Cost Path. Given an $m \times n$ grid where each cell contains a non-negative cost, find the path from the top-left to the bottom-right corner that minimizes the total cost. You can only move right or down. Return the minimum cost.

23.9 Exercises

Table 23.1: Memoization vs Tabulation Comparison

Criterion	Top-Down Memoization	Bottom-Up Tabulation
Approach	Recursive + cache	Iterative + table
Stack Usage	$O(n)$ Call Stack depth	$O(1)$ stack (no recursion)
Subproblems Solved	Only those needed	All subproblems 0 to n
Implementation	Often more intuitive	Often faster in practice
Space Optimization	Harder to apply	Easy with rolling variables

E1. [Level 1] Implement memoized Fibonacci and verify that `fib(40)` returns instantly.
E2. [Level 1] Solve the Climbing Stairs problem for $n = 30$ using tabulation.
E3. [Level 2] Solve the Coin Change problem: find the minimum coins for amount 67 using denominations [1, 5, 10, 25].
E4. [Level 2] Compute the number of unique grid paths for a 5×5 grid.
E5. [Level 3] Implement the Longest Common Subsequence and reconstruct the actual subsequence string (not just its length).
E6. [Level 4] Solve the 0/1 Knapsack Problem: given items with weights and values, maximize total value within a weight capacity using a 2D DP table.

23.10 Quiz

Q1. What are the two prerequisites for a problem to be solvable with Dynamic Programming?
Q2. What is the key difference between memoization and tabulation?
Q3. Why does naive recursive Fibonacci have $O(2^n)$ time complexity?
Q4. How can you reduce the space complexity of a 1D DP table from $O(n)$ to $O(1)$?
Q5. What is the time complexity of the Coin Change problem with n denominations and target amount A?

CHAPTER SUMMARY

- Dynamic Programming transforms exponential recursion into polynomial-time solutions.
- Top-Down Memoization caches recursive results; Bottom-Up Tabulation fills tables iteratively.
- Rolling variables optimize 1D DP space from $O(n)$ to $O(1)$.
- Classic DP problems include Fibonacci, Climbing Stairs, Coin Change, Grid Paths, and LCS.

INTERVIEW CORNER

Q: How do you approach a new DP problem in an interview? — Follow a 4-step framework: (1) **Define the state**: What does `dp[i]` represent? (2) **Find the recurrence**: How does `dp[i]` relate to smaller subproblems? (3) **Identify base cases**: What are the trivial solutions? (4) **Determine iteration order**: Ensure dependencies are computed before they are needed.

Q: When should you use memoization vs tabulation? — Use memoization when only a subset of subproblems need solving (sparse dependency graph). Use tabulation when all subproblems must be solved and you want to avoid stack overflow risks. Tabulation is also easier to space-optimize.

24 MODULE

Functional Programming

Functional Programming (FP) is a programming paradigm that treats computation as the evaluation of **mathematical functions** rather than sequences of imperative instructions. In functional code, data flows through **pure functions** that never mutate their inputs, never produce side effects, and always return the same output for the same input. JavaScript is a multi-paradigm language that supports functional programming alongside object-oriented and imperative styles, and modern JavaScript codebases increasingly leverage FP principles for cleaner, more testable, and more predictable code.

LEARNING OBJECTIVES

- Understand the core FP principles: pure functions, immutability, declarative style
- Write and identify pure functions vs impure functions
- Use immutable data patterns: spread syntax, `Object.freeze()`, non-mutating array methods
- Implement function composition: combining small functions into complex data pipelines
- Understand closures as a functional pattern for data encapsulation
- Implement currying: transforming multi-argument functions into chainable single-argument functions
- Apply `map`, `filter`, `reduce` as core functional data transformation tools
- Understand referential transparency and why it enables safe refactoring

24.1 Pure Functions

A **pure function** satisfies two invariants:

1. **Deterministic**: Given the same inputs, it always returns the same output.
2. **No Side Effects**: It does not modify external state, mutate arguments, perform I/O, or depend on anything other than its parameters.

```
// PURE: Depends only on inputs, returns new value, no mutation
function calculateDiscount(price, discountPercent) {
  return price - (price * discountPercent / 100);
}

console.log(calculateDiscount(100, 15)); // Always 85
console.log(calculateDiscount(100, 15)); // Always 85 (deterministic)

// IMPURE: Modifies external state (side effect!)
let totalRevenue = 0; // External mutable state

function recordSale(amount) {
```

```
  totalRevenue += amount; // Side effect: mutating external variable
  console.log(`Sale: $${amount}`); // Side effect: I/O
  return totalRevenue;
}

recordSale(50); // Returns 50, totalRevenue = 50
recordSale(50); // Returns 100, totalRevenue = 100 (not deterministic!)
```

Code Listing 24.1: Pure vs Impure Functions

BEST PRACTICE

Pure functions are inherently **testable** (no mocking required), **cacheable** (memoization works perfectly), **parallelizable** (no shared state conflicts), and **composable** (outputs feed directly into other functions). Maximize the percentage of pure functions in your codebase and isolate side effects at the boundaries.

24.2 Immutability: Never Mutate, Always Copy

In FP, data is **immutable**. Instead of modifying existing objects or arrays, you create **new copies** with the desired changes:

```
// MUTATING (bad in FP): Modifies original array
const numbers = [1, 2, 3, 4, 5];
// numbers.push(6); // Mutates the original array!

// IMMUTABLE (good in FP): Creates a new array
const withSix = [...numbers, 6]; // numbers is unchanged
console.log(numbers);   // [1, 2, 3, 4, 5] (original preserved)
console.log(withSix);   // [1, 2, 3, 4, 5, 6] (new array)

// Immutable object updates with spread
const fediProfile = { name: "Fedi", role: "developer", level: "senior" };
const promoted = { ...fediProfile, role: "architect" }; // New object, original unchanged

// Deep freeze to prevent accidental mutation
const config = Object.freeze({
  apiUrl: "https://api.fediapp.dev",
  timeout: 5000,
  retries: 3
});
// config.timeout = 10000; // Silently fails (or throws in strict mode)
```

Code Listing 24.2: Immutable Data Patterns in JavaScript

COMMON MISTAKE

`Object.freeze()` is **shallow**. Nested objects inside a frozen object can still be mutated. For deep immutability, use `structuredClone()` to create independent deep copies, or use libraries like Immer.

24.3 Function Composition

Function composition combines multiple small, focused functions into a data transformation pipeline. Each function's output becomes the next function's input:

```
// Small, focused, reusable pure functions
const trim = (str) => str.trim();
const toLower = (str) => str.toLowerCase();
const replaceSpaces = (str) => str.replace(/\s+/g, "-");
const removeSpecialChars = (str) => str.replace(/[^a-z0-9-]/g, "");

// Manual composition: right to left (innermost runs first)
function createSlug(title) {
  return removeSpecialChars(replaceSpaces(toLower(trim(title))));
}

// Utility: pipe (left to right, more readable)
function pipe(...fns) {
  return (value) => fns.reduce((acc, fn) => fn(acc), value);
}

// Create slug using pipe (reads top-to-bottom like a pipeline)
const slugify = pipe(trim, toLower, replaceSpaces, removeSpecialChars);

console.log(slugify("  Fedi's JavaScript Book!  ")); // "fedis-javascript-book"
console.log(slugify(" Dynamic Programming 101 ")); // "dynamic-programming-101"
```

Code Listing 24.3: Building a Data Pipeline with Composition

```
// compose: same as pipe but executes right-to-left (mathematical convention)
function compose(...fns) {
  return (value) => fns.reduceRight((acc, fn) => fn(acc), value);
}

const formatPrice = compose(
  (str) => `$${str}`,         // 3. Add dollar sign
  (num) => num.toFixed(2),    // 2. Format to 2 decimal places
  (price) => price * 0.9      // 1. Apply 10% discount (runs first!)
);

console.log(formatPrice(100)); // "$90.00"
```

Code Listing 24.4: Compose Utility (Right-to-Left Execution)

24.4 Currying

Currying transforms a function that takes multiple arguments into a sequence of functions that each take a single argument. This enables **partial application** — pre-filling some arguments to create specialized functions:

```
// Non-curried: takes all arguments at once
function multiply(a, b) {
  return a * b;
}

// Curried: returns a function chain
function curriedMultiply(a) {
  return function(b) {
    return a * b;
  };
}

// Arrow syntax (more concise):
const multiply2 = (a) => (b) => a * b;

// Create specialized functions via partial application
const double = multiply2(2);
const triple = multiply2(3);
const toPercent = multiply2(100);

console.log(double(5));     // 10
console.log(triple(5));     // 15
console.log(toPercent(0.85)); // 85

// Practical: Curried event logger
const logEvent = (category) => (action) => (label) => {
  console.log(`[${category}] ${action}: ${label}`);
};

const logUI = logEvent("UI");        // Partially applied: category = "UI"
const logClick = logUI("click");     // Further applied: action = "click"
logClick("signup-button");           // [UI] click: signup-button
logClick("nav-menu");                // [UI] click: nav-menu
```

Code Listing 24.5: Currying for Reusable Function Factories

24.5 Declarative vs Imperative Style

```
const transactions = [
  { user: "Fedi", amount: 120, type: "income" },
  { user: "Fedi", amount: 45, type: "expense" },
  { user: "Sam", amount: 200, type: "income" },
  { user: "Fedi", amount: 80, type: "expense" },
  { user: "Sam", amount: 30, type: "expense" }
];

// IMPERATIVE: HOW to process (step-by-step mutation)
let fediExpenseTotal = 0;
for (let i = 0; i < transactions.length; i++) {
  if (transactions[i].user === "Fedi" && transactions[i].type === "expense") {
    fediExpenseTotal += transactions[i].amount;
  }
```

```
}

// DECLARATIVE: WHAT to compute (data pipeline, no mutation)
const fediExpenses = transactions
  .filter(t => t.user === "Fedi" && t.type === "expense")
  .map(t => t.amount)
  .reduce((sum, amount) => sum + amount, 0);

console.log(fediExpenses); // 125 (both approaches give same result)
```

Code Listing 24.6: Imperative vs Declarative Data Processing

24.6 Closures as Functional Data Encapsulation

```
function createCounter(initialValue = 0) {
  let count = initialValue; // Enclosed state (private via closure)

  return {
    increment: () => ++count,
    decrement: () => --count,
    getCount: () => count,
    reset: () => { count = initialValue; return count; }
  };
}

const pageViews = createCounter();
pageViews.increment();
pageViews.increment();
pageViews.increment();
console.log(pageViews.getCount()); // 3
// console.log(count); // ReferenceError: count is private!
```

Code Listing 24.7: Closure-Based Counter (No Classes Needed)

MINI CHALLENGE

The FP Data Pipeline. Given an array of product objects with `name`, `price`, and `category` fields, build a purely functional pipeline using only `map`, `filter`, and `reduce` that: (1) filters products under $50, (2) applies a 10% tax to each price, (3) groups results by category into an object, (4) returns a Map of category to total revenue. No `for` loops or mutation allowed.

24.7 Exercises

E1. [Level 1] Rewrite an imperative loop that sums an array into a single `reduce()` call.
E2. [Level 1] Identify which of 5 given functions are pure and which are impure, and explain why.
E3. [Level 2] Implement a `pipe(...fns)` utility and use it to build a text processing pipeline.
E4. [Level 2] Implement an immutable array update helper `updateAt(arr, index, newValue)` using spread syntax.
E5. [Level 3] Implement a generic `curry(fn)` utility that auto-curries any function of any arity.

E6. [Level 4] Build a mini functional reactive library with `createSignal(value)` that returns `[getter, setter]` and supports `derived()` computed values.

24.8 Quiz

Q1. What two properties define a pure function?
Q2. Why is `Object.freeze()` considered "shallow" immutability?
Q3. What is the difference between `pipe` and `compose`?
Q4. What is currying, and how does it enable partial application?
Q5. Why are pure functions easier to test than impure functions?

CHAPTER SUMMARY

- Pure functions are deterministic and side-effect-free — the building blocks of FP.
- Immutability means creating new copies instead of mutating existing data structures.
- Function composition (`pipe`, `compose`) chains small functions into powerful data pipelines.
- Currying creates reusable, partially-applied function factories from multi-argument functions.

INTERVIEW CORNER

Q: What is referential transparency? — A function call is referentially transparent if it can be replaced with its return value without changing the program's behavior. This is only possible for pure functions. For example, `add(2, 3)` can always be replaced with `5`. This property enables safe refactoring, caching, and parallel execution.

Q: How do you handle side effects in a functional codebase? — Isolate side effects at the boundaries of your application (API calls, DOM updates, logging) and keep the core business logic pure. Use patterns like dependency injection, the Reader monad, or frameworks like Redux where reducers are pure and side effects are handled by middleware.

25 MODULE

Asynchronous JavaScript

JavaScript is single-threaded: it has one Call Stack and can execute only one line of code at any given moment. Yet modern web applications smoothly fetch network data, stream 4K video, animate graphics at 120 FPS, and respond to user clicks without freezing. How is this possible? The answer is the **JavaScript Event Loop** and its **Asynchronous Concurrency Model**. In this final curriculum module, you will master the engine's inner workings, Promises, `async`/`await`, network communication with `fetch()`, cancellation, and concurrent Promise combinators.

LEARNING OBJECTIVES

- Understand JavaScript's single-threaded concurrency model and thread blocking
- Trace the lifecycle of the Call Stack, Web APIs, Task Queue, and Microtask Queue
- Understand the evolution of asynchronous patterns from callbacks to Promises and `async`/`await`
- Fetch data from REST APIs using `fetch()`, including headers, POST payloads, and error codes
- Cancel active requests using the modern `AbortController` and `AbortSignal` APIs
- Master Promise combinators: `Promise.all`, `Promise.allSettled`, `Promise.race`, and `Promise.any`
- Implement an asynchronous task queue with concurrency limits from scratch

25.1 The Event Loop Architecture

The JavaScript runtime consists of four interconnected subsystems:

1. **Call Stack**: Executes synchronous JavaScript code one frame at a time.
2. **Web APIs / Node APIs**: Background threads handling timers, network requests, and DOM events.
3. **Microtask Queue**: High-priority queue for Promise callbacks and `queueMicrotask`.
4. **Task Queue (Macrotask Queue)**: Standard queue for `setTimeout`, `setInterval`, and I/O.

NOTE

The Event Loop Rule: Whenever the Call Stack becomes completely empty, the Event Loop first empties the **Microtask Queue** completely before taking the next item from the **Task Queue**. Additionally, rendering updates occur immediately after the Microtask Queue is cleared.

25.2 From Callbacks to Promises

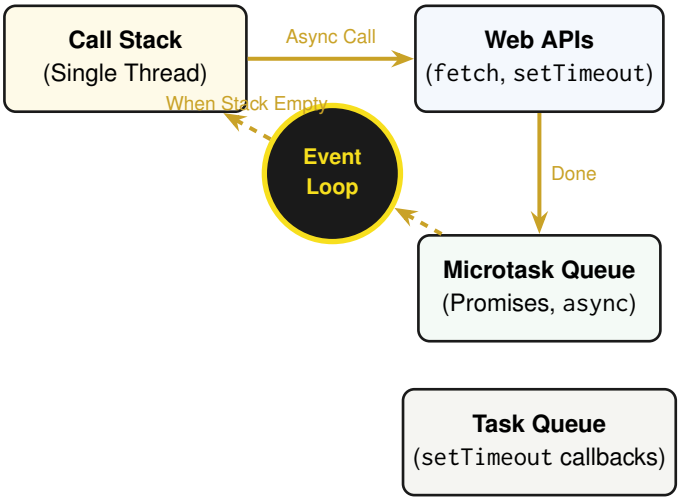


Figure 25.1: The JavaScript Event Loop and Concurrency Lifecycle

Early asynchronous JavaScript relied on nested callbacks, creating unreadable "Callback Hell" (the Pyramid of Doom). ES6 introduced **Promises** to model future values.

A Promise is in one of three states:

- **Pending**: Initial state, operation still in progress.
- **Fulfilled**: Operation completed successfully (`resolve(value)`).
- **Rejected**: Operation failed (`reject(error)`).

COMMON MISTAKE

The `new Promise()` executor function runs **synchronously**! Only the callbacks registered with `.then()` or `.catch()` are placed in the Microtask Queue to run asynchronously.

```
function fetchFediUserData(userId) {
  console.log("1. Inside executor (runs synchronously!)");
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (userId > 0) {
        resolve({ id: userId, username: "FediUser", role: "admin" });
      } else {
        reject(new Error("Invalid User ID"));
      }
    }, 500);
  });
}

console.log("2. Initiating request...");
fetchFediUserData(101)
  .then(user => console.log(`4. Welcome, ${user.username}!`))
  .catch(err => console.error("Error:", err.message))
  .finally(() => console.log("5. Request cycle completed."));
console.log("3. Continuing execution after request hook.");
```

Code Listing 25.1: Creating and consuming a Promise

25.3 Modern async and await

Introduced in ES2017, `async` and `await` provide synchronous-looking syntax for consuming Promises, eliminating chained `.then()` callbacks:

```
// An async function ALWAYS returns a Promise implicitly!
async function loadDashboard() {
  try {
    console.log("Loading dashboard data...");
    const user = await fetchFediUserData(101); // Pauses until Promise resolves
    console.log(`Loaded user: ${user.username} (${user.role})`);
    return user;
  } catch (error) {
    console.error("Dashboard failed to load:", error.message);
  }
}

loadDashboard();
```

Code Listing 25.2: Async / Await Syntax

25.4 The Fetch API: HTTP Communication

The `fetch()` API communicates with backend REST services. It accepts a resource URL and an options object containing methods, headers, and payload bodies:

```
async function createFediUser(profileData) {
  const response = await fetch("https://api.fediapp.dev/v2/users", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "Authorization": "Bearer TOKEN_HERE"
    },
    body: JSON.stringify(profileData)
  });

  // fetch() only rejects on network failures, not on HTTP error codes (like 404 or 500)
  if (!response.ok) {
    throw new Error(`HTTP Error: ${response.status} ${response.statusText}`);
  }

  const data = await response.json();
  return data;
}
```

Code Listing 25.3: Sending POST Request with fetch()

25.5 Cancelling Requests with AbortController

To cancel network requests (e.g., when a user clicks away from a page or to implement timeouts), use `AbortController`:

```
async function fetchWithTimeout(url, timeoutMs = 3000) {
  const controller = new AbortController();
  const signal = controller.signal;

  // Set timeout to abort request
  const timer = setTimeout(() => controller.abort(), timeoutMs);

  try {
    const response = await fetch(url, { signal });
    clearTimeout(timer); // Clear timer if request succeeds
    if (!response.ok) throw new Error("HTTP Error");
    return await response.json();
  } catch (error) {
    clearTimeout(timer);
    if (error.name === "AbortError") {
      throw new Error("Request timed out after " + timeoutMs + "ms");
    }
    throw error;
  }
}
```

Code Listing 25.4: Implementing Network Timeouts with AbortController

25.6 Concurrent Promise Combinators

Table 25.1: Promise Combinator Operations

Combinator	Resolves When	Rejects When
`Promise.all()`	**All** resolve	**Any** rejects (fails fast)
`Promise.allSettled()`	**All** settle (success/fail)	Never rejects
`Promise.race()`	**First** resolves or rejects	First resolves or rejects
`Promise.any()`	**First** resolves	**All** reject (returns AggregateError)

```
const p1 = fetch("/api/user").then(r => r.json());
const p2 = fetch("/api/notifications").then(r => r.json());
const p3 = fetch("/api/messages").then(r => r.json());

// 1. Promise.all: Runs all concurrently; FAILS FAST if ANY single promise rejects
const [user, notifications, messages] = await Promise.all([p1, p2, p3]);

// 2. Promise.allSettled: Waits for ALL promises to finish, never rejects
const results = await Promise.allSettled([p1, p2, p3]);
results.forEach(res => {
  if (res.status === "fulfilled") {
    console.log("Data:", res.value);
```

```
  } else {
    console.warn("Failed:", res.reason);
  }
});
```

Code Listing 25.5: Parallel Execution with Promise Combinators

MINI CHALLENGE

The Resilient Fetch with Timeout and Retries. Write an asynchronous function `fetchWithRetry(url, maxRetries = 3, timeoutMs = 3000)` that uses `AbortController` to cancel requests exceeding the timeout, and automatically retries up to 3 times before throwing.

25.7 Exercises

E1. [Level 1] Write a helper `delay(ms)` that returns a Promise resolving after ms milliseconds using `setTimeout`.

E2. [Level 1] Convert a Promise chain using `.then().catch()` into an equivalent `async/await` function with `try/catch`.

E3. [Level 2] Fetch user data from `jsonplaceholder.typicode.com/users` and display their names in the DOM.

E4. [Level 2] Implement `Promise.race()` to create a network request that times out if it takes longer than 2 seconds.

E5. [Level 3] Implement a custom version of `myPromiseAll(promises)` from scratch that replicates native `Promise.all`.

E6. [Level 4] Build an asynchronous Task Queue with a concurrency limiter that executes at most K network tasks concurrently.

25.8 Quiz

Q1. Is JavaScript single-threaded or multi-threaded? How does it handle asynchronous I/O?

Q2. What is the difference between the Microtask Queue and the Macrotask (Task) Queue?

Q3. Why does `fetch()` not reject on HTTP 404 or 500 error status codes?

Q4. What is the advantage of `Promise.allSettled()` over `Promise.all()`?

Q5. What happens if you forget to place an `await` keyword before a Promise-returning function?

CHAPTER SUMMARY

- The Event Loop enables non-blocking asynchronous execution on a single-threaded runtime.
- Microtasks (Promises) always take priority over Macrotasks (`setTimeout`).
- Use `async/await` and `try...catch` for clean, synchronous-looking asynchronous logic.
- Run independent asynchronous operations in parallel with `Promise.all` and `Promise.allSettled`.

INTERVIEW CORNER

Q: What is the output order of this script?

```
console.log(1); setTimeout(() => console.log(2), 0); Promise.resolve().then(() => console.log(3)
```

Answer: `1, 4, 3, 2`. Synchronous code runs first (`1, 4`). Next, the Event Loop empties the Microtask Queue (`3`). Finally, the Task Queue runs `setTimeout` (`2`).

Q: Why is blocking the main thread bad, and how do you execute heavy computation off-thread? — Blocking the main thread stops the Event Loop from processing user clicks, scroll events, and animations, making the UI appear frozen. For heavy math or sorting operations, offload the workload using **Web Workers** which run on separate OS threads and communicate with the main script via asynchronous message passing.

PART V

Build with Fedi — Capstone Projects

CAPSTONE PROJECTS

Twelve complete projects, from a friendly greeting bot to a visual pathfinding engine. Each one is real, buildable, and designed to cement a module of the book. Build them all — that is how beginners become masters.

26 MODULE

Project 1: Fedi Greeting Bot

The **Fedi Greeting Bot** is your first complete interactive project. It brings together all core concepts from Tier 1: variables, template literals, conditional logic, string methods, and DOM interaction. The bot asks for user details, analyzes time of day, calculates personalized status badges, and generates lively contextual greetings.

26.1 Project Overview and Goals

LEARNING OBJECTIVES

- **Goal**: Build a reactive greeting generator that responds to user input, time of day, and selected mood.
- **Concepts Applied**: `const/let`, template literals, string manipulation, `querySelector`, and event listeners.
- **Features**: Automatic time period detection, title-casing algorithms, and personalized status badges.

26.2 User Interface Architecture and HTML

The user interface consists of a card container with an avatar header, input fields, a mood selector, and an output display card:

```
<div class="card greeting-bot-card">
  <div class="card-header">
    <span class="bot-icon">[BOT]</span>
    <h2>Fedi Greeting Bot</h2>
  </div>

  <div class="form-group">
    <label for="userNameInput">Your Name:</label>
    <input type="text" id="userNameInput" placeholder="e.g. Fedi" autocomplete="off" />
  </div>

  <div class="form-group">
    <label for="moodSelect">Current Mood:</label>
    <select id="moodSelect">
      <option value="excited">[POW] Excited to code</option>
      <option value="focused">[AIM] Deep Focus</option>
      <option value="chill">[ZEN] Relaxed and Calm</option>
```

```
    </select>
  </div>

  <button id="generateGreetingBtn" class="btn btn-primary">Generate Greeting</button>

  <div id="greetingOutput" class="greeting-display" hidden></div>
</div>
```

Code Listing 26.1: Greeting Bot HTML Structure

26.3 Styling the Component with CSS

```
.greeting-bot-card {
  max-width: 480px;
  margin: 2rem auto;
  padding: 1.5rem;
  background: #ffffff;
  border: 1px solid #d8d8ce;
  border-radius: 8px;
  box-shadow: 0 4px 12px rgba(0, 0, 0, 0.05);
}

.bot-icon {
  font-weight: bold;
  color: #C9A227;
  margin-right: 0.5rem;
}

.greeting-display {
  margin-top: 1.5rem;
  padding: 1.2rem;
  background: #f7f7f2;
  border-left: 4px solid #C9A227;
  border-radius: 4px;
}

.badge {
  display: inline-block;
  padding: 0.25rem 0.6rem;
  background: #1A1A1A;
  color: #F7DF1E;
  border-radius: 4px;
  font-size: 0.85rem;
  font-weight: bold;
}
```

Code Listing 26.2: Greeting Bot CSS Stylesheet

26.4 JavaScript Implementation and Logic

```
// 1. Cache DOM Elements
const nameInput = document.querySelector("#userNameInput");
const moodSelect = document.querySelector("#moodSelect");
const generateBtn = document.querySelector("#generateGreetingBtn");
const outputDiv = document.querySelector("#greetingOutput");

// 2. Pure Helper Functions
function getTimePeriod() {
  const currentHour = new Date().getHours();
  if (currentHour >= 5 && currentHour < 12) {
    return "Good morning";
  } else if (currentHour >= 12 && currentHour < 18) {
    return "Good afternoon";
  } else {
    return "Good evening";
  }
}

function formatName(rawName) {
  const trimmed = rawName.trim();
  if (trimmed.length === 0) return "Fellow Developer";
  // Capitalize first letter, lowercase remaining
  return trimmed[0].toUpperCase() + trimmed.slice(1).toLowerCase();
}

function getMoodBadge(moodKey) {
  switch (moodKey) {
    case "excited": return "[POWER] High Energy Developer";
    case "focused": return "[FOCUS] Deep Flow State";
    case "chill":   return "[ZEN] Calm Problem Solver";
    default:        return "[ACTIVE] Ready to Build";
  }
}

// 3. Controller Action
function renderGreeting() {
  const userName = formatName(nameInput.value);
  const timeGreeting = getTimePeriod();
  const badgeLabel = getMoodBadge(moodSelect.value);

  // Construct message using template literals
  outputDiv.innerHTML = `
    <div class="greeting-box">
      <h3>${timeGreeting}, ${userName}!</h3>
      <p>Welcome to <strong>FediApp</strong>. Your current status is:</p>
      <div style="margin: 0.8rem 0;"><span class="badge">${badgeLabel}</span></div>
      <p class="tagline">"Every line of JavaScript you write brings you closer to mastery
    ↪ ."</p>
    </div>
  `;
  outputDiv.hidden = false;
}

// 4. Event Listeners
generateBtn.addEventListener("click", renderGreeting);
nameInput.addEventListener("keydown", (e) => {
```

```
  if (e.key === "Enter") renderGreeting();
});
```

Code Listing 26.3: Greeting Bot JavaScript Engine

26.5 Step-by-Step Code Analysis

1. **Input Sanitization**: `nameInput.value.trim()` strips hazardous leading/trailing whitespace before formatting.
2. **Time Determination**: `new Date().getHours()` returns integer $0 \ldots 23$, allowing accurate period segmentation.
3. **Keyboard Ergonomics**: Binding `keydown` on the text input allows users to submit via the Enter key without reaching for the mouse.

26.6 Testing Checklist and Extensions

- **Empty Input Test**: Pressing Generate with blank input displays "Fellow Developer".
- **Case Test**: Typing "fEdI" renders formatted title-cased "Fedi".
- **Extension 1**: Add an internationalization dropdown supporting French ("Bonjour"), Spanish ("Hola"), and Arabic ("Marhaban").
- **Extension 2**: Calculate and display the user's local timezone offset using `Intl.DateTimeFormat().resolvedOptic`

27 MODULE

Project 2: Fedi Digital Pet Game

The **Fedi Digital Pet Game** is a real-time virtual simulation that cements your understanding of state modeling, objects, setInterval game loops, arithmetic clamping, and dynamic progress bar updates. You will manage the vital statistics of a digital companion named **FediBot**, preventing its hunger and fatigue from reaching critical limits while maintaining high happiness levels.

27.1 Project Requirements and State Machine

LEARNING OBJECTIVES

- **Goal**: Build a real-time stateful virtual pet simulation with decaying stats and action buttons.
- **Concepts Applied**: State objects, methods, `setInterval`, arithmetic boundaries, and DOM progress meters.
- **Features**: Periodic game loop, happiness/hunger tracking, visual ASCII mood avatars, and death guard conditions.

27.2 HTML Layout Structure

```
<div class="pet-game-card">
  <header class="pet-header">
    <h2 id="petName">FediBot</h2>
    <div id="petAvatar" class="avatar-display">[ HAPPY ^_^ ]</div>
  </header>

  <div class="status-meters">
    <div class="meter-group">
      <label>Hunger: <span id="hungerVal">40</span>%</label>
      <progress id="hungerBar" value="40" max="100"></progress>
    </div>
    <div class="meter-group">
      <label>Happiness: <span id="happinessVal">80</span>%</label>
      <progress id="happinessBar" value="80" max="100"></progress>
    </div>
    <div class="meter-group">
      <label>Energy: <span id="energyVal">100</span>%</label>
      <progress id="energyBar" value="100" max="100"></progress>
    </div>
  </div>
```

```

  <div class="action-toolbar">
    <button id="feedBtn" class="btn btn-action">[+] Feed Snack</button>
    <button id="playBtn" class="btn btn-action">[*] Play Catch</button>
    <button id="sleepBtn" class="btn btn-action">[z] Rest and Sleep</button>
  </div>

  <div id="gameLog" class="status-log-box">FediBot is ready to play!</div>
</div>
```

Code Listing 27.1: Digital Pet Game HTML Markup

27.3 CSS Component Styling

```
.pet-game-card {
  max-width: 440px;
  margin: 2rem auto;
  padding: 1.5rem;
  background: #ffffff;
  border: 1px solid #d8d8ce;
  border-radius: 8px;
  text-align: center;
}

.avatar-display {
  font-family: monospace;
  font-size: 1.8rem;
  font-weight: bold;
  padding: 1.2rem;
  background: #f4f4ef;
  border-radius: 6px;
  margin: 1rem 0;
  color: #1A1A1A;
}

.meter-group {
  margin: 0.8rem 0;
  text-align: left;
}

.meter-group label {
  display: block;
  font-size: 0.9rem;
  margin-bottom: 0.2rem;
}

progress {
  width: 100%;
  height: 12px;
  border-radius: 6px;
}

.action-toolbar {
```

```
  display: flex;
  gap: 0.5rem;
  justify-content: center;
  margin-top: 1.2rem;
}
```

Code Listing 27.2: Digital Pet CSS Styles

27.4 JavaScript State and Game Engine

```
const pet = {
  name: "FediBot",
  hunger: 40,      // 0 = full, 100 = starving
  happiness: 80,   // 0 = sad, 100 = ecstatic
  energy: 100,     // 0 = exhausted, 100 = energetic
  isAlive: true,

  clamp(val, min = 0, max = 100) {
    return Math.max(min, Math.min(max, val));
  },

  feed() {
    if (!this.isAlive) return;
    this.hunger = this.clamp(this.hunger - 25);
    this.happiness = this.clamp(this.happiness + 5);
    this.render("Ate a delicious snack! Hunger decreased.");
  },

  play() {
    if (!this.isAlive) return;
    if (this.energy < 20) {
      this.render("Too exhausted to play! FediBot needs sleep.");
      return;
    }
    this.happiness = this.clamp(this.happiness + 20);
    this.hunger = this.clamp(this.hunger + 15);
    this.energy = this.clamp(this.energy - 25);
    this.render("Played a fun game! Happiness boosted.");
  },

  sleep() {
    if (!this.isAlive) return;
    this.energy = 100;
    this.hunger = this.clamp(this.hunger + 20);
    this.render("Had a refreshing nap! Energy fully restored.");
  },

  tick() {
    if (!this.isAlive) return;
    this.hunger = this.clamp(this.hunger + 5);
    this.happiness = this.clamp(this.happiness - 4);
    this.energy = this.clamp(this.energy - 2);

```

```
    if (this.hunger >= 100 || this.happiness <= 0) {
      this.isAlive = false;
      this.render("FediBot has fallen asleep permanently. Game Over!");
    } else {
      this.render();
    }
  },

  render(message = "") {
    document.querySelector("#hungerVal").textContent = this.hunger;
    document.querySelector("#hungerBar").value = this.hunger;
    document.querySelector("#happinessVal").textContent = this.happiness;
    document.querySelector("#happinessBar").value = this.happiness;
    document.querySelector("#energyVal").textContent = this.energy;
    document.querySelector("#energyBar").value = this.energy;

    const avatar = document.querySelector("#petAvatar");
    if (!this.isAlive) {
      avatar.textContent = "[ GAME OVER X_X ]";
    } else if (this.hunger > 75) {
      avatar.textContent = "[ HUNGRY >_< ]";
    } else if (this.happiness > 70) {
      avatar.textContent = "[ HAPPY ^_^ ]";
    } else {
      avatar.textContent = "[ NEUTRAL ._. ]";
    }

    if (message) {
      document.querySelector("#gameLog").textContent = message;
    }
  }
};

// Event Bindings
document.querySelector("#feedBtn").addEventListener("click", () => pet.feed());
document.querySelector("#playBtn").addEventListener("click", () => pet.play());
document.querySelector("#sleepBtn").addEventListener("click", () => pet.sleep());

// Start Game Loop (ticks every 3000ms)
const gameTimer = setInterval(() => pet.tick(), 3000);
```

Code Listing 27.3: Digital Pet Simulation Engine

27.5 Testing Checklist and Extensions

- **Boundary Testing**: Click Feed repeatedly; verify that hunger never drops below 0%.
- **Starvation Test**: Leave the tab running; verify that hunger reaches 100% and game-over triggers cleanly.
- **Extension 1**: Store pet state in `localStorage` so the pet survives page reloads.
- **Extension 2**: Add sound effects for feeding and playing using the Web Audio API.

28 MODULE

Project 3: Fedi Flashcard Quiz App

The **Fedi Flashcard Quiz App** is an interactive study application designed to help developers master JavaScript syntax and concepts. It features 3D flip card animations, randomized question shuffling, category tags, and real-time score tracking. This project reinforces array operations, object collections, and CSS transitions.

28.1 Project Overview and Requirements

LEARNING OBJECTIVES

- **Goal**: Build an interactive study app with flipping flashcards, score tracking, and question progress.
- **Concepts Applied**: Arrays of objects, array shuffling, event listeners, and dynamic DOM rendering.
- **Features**: 3D flip animation, category tags, mastery scoring, and final summary screen.

28.2 HTML Layout Structure

```
<div class="quiz-container">
  <header class="quiz-header">
    <h2>Fedi JS Mastery Flashcards</h2>
    <div class="score-tracker">Score: <span id="scoreDisplay">0</span> / <span id="
    ↪ totalDisplay">0</span></div>
  </header>

  <div class="flashcard-scene">
    <div id="activeFlashcard" class="flashcard">
      <div class="card-face card-front">
        <span id="cardCategory" class="badge">Scope</span>
        <h3 id="cardQuestion">Question loading...</h3>
        <p class="hint">Click card to reveal answer</p>
      </div>
      <div class="card-face card-back">
        <span class="badge">Answer</span>
        <p id="cardAnswer">Answer loading...</p>
        <div class="card-actions">
          <button id="gotWrongBtn" class="btn btn-danger">[x] Need Practice</button>
          <button id="gotRightBtn" class="btn btn-success">[ok] Got It Right</button>
        </div>
```

```
      </div>
    </div>
  </div>

  <div class="quiz-controls">
    <button id="prevBtn" class="btn">[<-] Previous</button>
    <span id="progressIndicator">Card 1 of 10</span>
    <button id="nextBtn" class="btn">Next [->]</button>
  </div>
</div>
```

Code Listing 28.1: Flashcard Quiz HTML Markup

28.3 CSS 3D Flip Stylesheet

```
.flashcard-scene {
  width: 100%;
  max-width: 520px;
  height: 280px;
  perspective: 1000px;
  margin: 1.5rem auto;
}

.flashcard {
  width: 100%;
  height: 100%;
  position: relative;
  transition: transform 0.6s cubic-bezier(0.4, 0, 0.2, 1);
  transform-style: preserve-3d;
  cursor: pointer;
}

.flashcard.is-flipped {
  transform: rotateY(180deg);
}

.card-face {
  position: absolute;
  width: 100%;
  height: 100%;
  backface-visibility: hidden;
  border-radius: 8px;
  padding: 1.5rem;
  box-sizing: border-box;
  display: flex;
  flex-direction: column;
  justify-content: space-between;
  border: 1px solid #d8d8ce;
  background: #ffffff;
}

.card-back {
  transform: rotateY(180deg);
```

```
  background: #f7f7f2;
}
```

Code Listing 28.2: Flashcard 3D Perspective CSS

28.4 JavaScript Flashcard Engine

```
const flashcards = [
  {
    category: "Variables",
    question: "What is the difference between const and let?",
    answer: "const creates a block-scoped binding that cannot be reassigned. let creates
    ↪ a block-scoped variable that can be reassigned."
  },
  {
    category: "Functions",
    question: "What is a closure in JavaScript?",
    answer: "A closure is a function bundled with references to its surrounding lexical
    ↪ environment, remembering outer variables even after the outer function returns."
  },
  {
    category: "Arrays",
    question: "How does slice() differ from splice()?",
    answer: "slice() returns a shallow copy of a portion of an array without modifying
    ↪ the original. splice() mutates the array in place."
  },
  {
    category: "Async",
    question: "What is the priority order between Microtasks and Macrotasks?",
    answer: "Microtasks (Promises) are always emptied completely by the Event Loop before
    ↪  the next Macrotask (setTimeout) is executed."
  }
];

class FlashcardQuiz {
  constructor(cards) {
    this.cards = cards;
    this.currentIndex = 0;
    this.score = 0;
    this.isFlipped = false;
    this.init();
  }

  init() {
    this.cardEl = document.querySelector("#activeFlashcard");
    this.cardEl.addEventListener("click", (e) => {
      if (!e.target.closest("button")) {
        this.toggleFlip();
      }
    });

    document.querySelector("#gotRightBtn").addEventListener("click", () => this.
    ↪ handleAnswer(true));
```

```
    document.querySelector("#gotWrongBtn").addEventListener("click", () => this.
    ↪ handleAnswer(false));
    document.querySelector("#nextBtn").addEventListener("click", () => this.nextCard());
    document.querySelector("#prevBtn").addEventListener("click", () => this.prevCard());

    this.render();
  }

  toggleFlip() {
    this.isFlipped = !this.isFlipped;
    this.cardEl.classList.toggle("is-flipped", this.isFlipped);
  }

  handleAnswer(isCorrect) {
    if (isCorrect) this.score++;
    this.nextCard();
  }

  nextCard() {
    if (this.currentIndex < this.cards.length - 1) {
      this.currentIndex++;
      this.isFlipped = false;
      this.cardEl.classList.remove("is-flipped");
      this.render();
    }
  }

  prevCard() {
    if (this.currentIndex > 0) {
      this.currentIndex--;
      this.isFlipped = false;
      this.cardEl.classList.remove("is-flipped");
      this.render();
    }
  }

  render() {
    const cur = this.cards[this.currentIndex];
    document.querySelector("#cardCategory").textContent = cur.category;
    document.querySelector("#cardQuestion").textContent = cur.question;
    document.querySelector("#cardAnswer").textContent = cur.answer;
    document.querySelector("#scoreDisplay").textContent = this.score;
    document.querySelector("#totalDisplay").textContent = this.cards.length;
    document.querySelector("#progressIndicator").textContent = `Card ${this.currentIndex +
    ↪  1} of ${this.cards.length}`;
  }
}

const quiz = new FlashcardQuiz(flashcards);
```

Code Listing 28.3: Flashcard Quiz JavaScript Engine

28.5 Testing Checklist and Extensions

- **Checklist**: Test 3D flipping, button event bubbling isolation, and boundary conditions.
- **Extension 1**: Add a Fisher-Yates shuffle method to randomize card order.
- **Extension 2**: Persist wrong cards to a "Needs Review" filtered sub-deck in `localStorage`.

29 MODULE

Project 4: Fedi Motorcycle Shop

The **Fedi Motorcycle Shop** is a commercial catalog and shopping cart application. It integrates arrays of objects, higher-order functions (`filter`, `map`, `reduce`, `sort`), dynamic search filtering, price range sorting, and a shopping cart with coupon discounts.

29.1 Project Requirements and Architecture

LEARNING OBJECTIVES

- **Goal**: Build an e-commerce inventory filter and cart manager with real-time calculations.
- **Concepts Applied**: Higher-Order Functions, array pipeline chaining, event delegation, and cart state.
- **Features**: Search bar, category filter pills, price sorting dropdown, and cart drawer.

29.2 HTML Catalog Layout

```
<div class="shop-container">
  <header class="shop-header">
    <h2>[MOTO] Fedi Moto Emporium</h2>
    <button id="viewCartBtn" class="btn btn-cart">[CART] Cart (<span id="cartCount">0</
    ↪ span>)</button>
  </header>

  <div class="controls-bar">
    <input type="text" id="searchInput" placeholder="Search models (e.g. Yamaha, Ninja)...
    ↪ " />
    <select id="categoryFilter">
      <option value="all">All Categories</option>
      <option value="Sport">Sport</option>
      <option value="Cruiser">Cruiser</option>
      <option value="Adventure">Adventure</option>
    </select>
    <select id="sortSelect">
      <option value="price-asc">Price: Low to High</option>
      <option value="price-desc">Price: High to Low</option>
    </select>
  </div>

  <div id="productGrid" class="grid-container"></div>
```

```
</div>
```

Code Listing 29.1: Motorcycle Shop Catalog HTML

29.3 CSS Grid Stylesheet

```
.shop-container {
  max-width: 960px;
  margin: 2rem auto;
  padding: 1rem;
}

.controls-bar {
  display: flex;
  gap: 1rem;
  margin: 1.5rem 0;
}

.grid-container {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(260px, 1fr));
  gap: 1.5rem;
}

.bike-card {
  padding: 1.2rem;
  background: #ffffff;
  border: 1px solid #d8d8ce;
  border-radius: 8px;
  display: flex;
  flex-direction: column;
  justify-content: space-between;
}

.price-tag {
  font-size: 1.3rem;
  font-weight: bold;
  color: #1E7A34;
  margin: 0.8rem 0;
}
```

Code Listing 29.2: Catalog Grid CSS

29.4 JavaScript Catalog and Cart Engine

```
const inventory = [
  { id: "MOTO-01", brand: "Yamaha", model: "MT-07", price: 8199, category: "Sport", cc: 6
    ↪ 89 },
  { id: "MOTO-02", brand: "Kawasaki", model: "Ninja ZX-6R", price: 11999, category: "
    ↪ Sport", cc: 636 },
```

```
  { id: "MOTO-03", brand: "Harley-Davidson", model: "Iron 883", price: 10499, category: "Cruiser", cc: 883 },
  { id: "MOTO-04", brand: "BMW", model: "R 1250 GS", price: 17995, category: "Adventure", cc: 1254 },
  { id: "MOTO-05", brand: "Ducati", model: "Monster Plus", price: 12995, category: "Sport", cc: 937 }
];

let cart = [];

function renderCatalog() {
  const query = document.querySelector("#searchInput").value.toLowerCase().trim();
  const category = document.querySelector("#categoryFilter").value;
  const sortOrder = document.querySelector("#sortSelect").value;

  // Higher-Order Function Data Pipeline
  const filtered = inventory
    .filter(bike => {
      const matchesSearch = bike.brand.toLowerCase().includes(query) || bike.model.toLowerCase().includes(query);
      const matchesCat = category === "all" || bike.category === category;
      return matchesSearch && matchesCat;
    })
    .sort((a, b) => {
      return sortOrder === "price-asc" ? a.price - b.price : b.price - a.price;
    });

  const grid = document.querySelector("#productGrid");
  grid.innerHTML = filtered.map(bike => `
    <div class="bike-card">
      <h3>${bike.brand} ${bike.model}</h3>
      <p class="specs">Category: <strong>${bike.category}</strong> | Engine: <strong>${bike.cc}cc</strong></p>
      <div class="price-tag">$${bike.price.toLocaleString()}</div>
      <button class="btn btn-primary add-to-cart" data-id="${bike.id}">Add to Cart</button>
    </div>
  `).join("");
}

// Event Delegation for Adding to Cart
document.querySelector("#productGrid").addEventListener("click", (e) => {
  if (e.target.matches("button.add-to-cart")) {
    const bikeId = e.target.dataset.id;
    const bike = inventory.find(b => b.id === bikeId);
    if (bike) {
      cart.push(bike);
      document.querySelector("#cartCount").textContent = cart.length;
    }
  }
});

// Setup Filter Listeners
document.querySelector("#searchInput").addEventListener("input", renderCatalog);
document.querySelector("#categoryFilter").addEventListener("change", renderCatalog);
document.querySelector("#sortSelect").addEventListener("change", renderCatalog);
```

```
54
55 // Initial Render
56 renderCatalog();
```

Code Listing 29.3: Motorcycle Shop JavaScript Logic

29.5 Testing Checklist and Extensions

- **Checklist**: Test real-time typing filtering, category switching, and sort ordering.
- **Extension 1**: Build a slide-out cart drawer that lists cart items with a remove button and computes sales tax and shipping.
- **Extension 2**: Add a coupon code field that applies a 10% discount when entering `"FEDI10"`.

30 MODULE

Project 5: Fedi Drum Machine

The **Fedi Drum Machine** is an interactive, low-latency musical soundboard application. It connects keyboard events and mouse clicks to audio playback, visually illuminating drum pads and allowing users to trigger beats or record custom rhythm sequences. This project reinforces DOM events, the HTMLMediaElement API, and active UI transitions.

30.1 Project Overview and Requirements

LEARNING OBJECTIVES

- **Goal**: Create a responsive 8-pad digital drum machine triggered via keyboard or click.
- **Concepts Applied**: Audio playback, `keydown` event listeners, keycode mapping, active class animations.
- **Features**: Zero-latency audio triggering, visual glow animations, and volume master fader.

30.2 HTML Pad Matrix Layout

```
<div class="drum-container">
  <h2>[STUDIO] Fedi Studio Beat Machine</h2>
  <div class="pad-grid">
    <div class="drum-pad" data-key="a">
      <kbd>A</kbd>
      <span class="sound">Kick</span>
      <audio src="assets/sounds/kick.wav" data-key="a"></audio>
    </div>
    <div class="drum-pad" data-key="s">
      <kbd>S</kbd>
      <span class="sound">Snare</span>
      <audio src="assets/sounds/snare.wav" data-key="s"></audio>
    </div>
    <div class="drum-pad" data-key="d">
      <kbd>D</kbd>
      <span class="sound">Hi-Hat</span>
      <audio src="assets/sounds/hihat.wav" data-key="d"></audio>
    </div>
    <div class="drum-pad" data-key="f">
      <kbd>F</kbd>
      <span class="sound">Clap</span>
      <audio src="assets/sounds/clap.wav" data-key="f"></audio>
```

```
    </div>
  </div>
</div>
```

Code Listing 30.1: Drum Machine HTML Matrix

30.3 CSS Pad Visualizer Stylesheet

```
.pad-grid {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  gap: 1rem;
  max-width: 560px;
  margin: 2rem auto;
}

.drum-pad {
  padding: 2rem 1rem;
  background: #1A1A1A;
  color: #ffffff;
  border-radius: 8px;
  text-align: center;
  cursor: pointer;
  transition: all 0.08s ease;
  user-select: none;
}

.drum-pad.playing {
  transform: scale(1.08);
  border-color: #F7DF1E;
  box-shadow: 0 0 16px rgba(247, 223, 30, 0.6);
  background: #2a2a2a;
}

kbd {
  display: block;
  font-size: 1.8rem;
  font-weight: bold;
  color: #F7DF1E;
}
```

Code Listing 30.2: Drum Pad Animation Styles

30.4 JavaScript Audio Controller

```
function playSound(keyLetter) {
  const audio = document.querySelector(`audio[data-key="${keyLetter.toLowerCase()}"]`);
  const pad = document.querySelector(`.drum-pad[data-key="${keyLetter.toLowerCase()}"]`);

  if (!audio || !pad) return;

```

```
  // Rewind to start to allow rapid consecutive hits with zero latency!
  audio.currentTime = 0;
  audio.play().catch(e => console.warn("Audio blocked:", e));

  // Add visual excitation glow class
  pad.classList.add("playing");
}

// Remove animation class after transition ends
const pads = document.querySelectorAll(".drum-pad");
pads.forEach(pad => {
  pad.addEventListener("transitionend", (e) => {
    if (e.propertyName === "transform") {
      pad.classList.remove("playing");
    }
  });

  // Click support
  pad.addEventListener("click", () => {
    playSound(pad.dataset.key);
  });
});

// Global Keyboard Listener
window.addEventListener("keydown", (e) => {
  playSound(e.key);
});
```

Code Listing 30.3: Drum Machine Audio Engine

30.5 Testing Checklist and Extensions

- **Checklist**: Press keys rapidly; verify that sounds restart instantly without audio clipping.
- **Extension 1**: Add a pattern recorder that saves timestamps of key presses and replays the rhythm.
- **Extension 2**: Add a BPM Metronome ticker that flashes on every beat.

31 MODULE

Project 6: Fedi Travel Weather Planner

The **Fedi Travel Weather Planner** is a modern asynchronous dashboard that allows travelers to search for cities, fetch real-time multi-day meteorological forecasts from a REST API, and render localized climate cards. This project cements your mastery of the `fetch()` API, `async/await`, JSON transformation, loading states, and error handling.

31.1 Project Overview and Requirements

LEARNING OBJECTIVES

- **Goal**: Build an asynchronous weather search engine with dynamic 5-day forecast cards.
- **Concepts Applied**: Asynchronous `fetch()`, `async/await`, `try...catch`, loading states, `Intl.DateTimeFormat`.
- **Features**: Search bar, Celsius/Fahrenheit conversion, weather condition displays, and error banners.

31.2 HTML Dashboard Structure

```
<div class="weather-dashboard">
  <header class="weather-header">
    <h2>[WEATHER] Fedi Weather and Travel Planner</h2>
    <form id="weatherSearchForm">
      <input type="text" id="cityInput" placeholder="Search city (e.g. Tunis, Tokyo,
    ↪ Paris)..." required />
      <button type="submit" class="btn btn-primary">Search</button>
    </form>
  </header>

  <div id="loadingSpinner" class="spinner" hidden>Loading forecast...</div>
  <div id="errorBanner" class="alert alert-danger" hidden></div>

  <main id="weatherContent" class="weather-content" hidden>
    <div class="current-weather">
      <h3 id="locationTitle">Tunis, TN</h3>
      <div id="currentTemp" class="temp-big">28 deg C</div>
      <p id="conditionText">Wind: 14 km/h</p>
    </div>
    <div id="forecastGrid" class="forecast-cards-grid"></div>
  </main>
```

```
</div>
```

Code Listing 31.1: Weather Planner HTML Dashboard

31.3 CSS Dashboard Layout

```
.weather-dashboard {
  max-width: 720px;
  margin: 2rem auto;
  padding: 1.5rem;
  background: #ffffff;
  border-radius: 8px;
  box-shadow: 0 4px 16px rgba(0, 0, 0, 0.08);
}

.temp-big {
  font-size: 3rem;
  font-weight: bold;
  color: #1A1A1A;
  margin: 0.5rem 0;
}

.forecast-cards-grid {
  display: grid;
  grid-template-columns: repeat(5, 1fr);
  gap: 0.8rem;
  margin-top: 1.5rem;
}

.forecast-day-card {
  padding: 0.8rem;
  text-align: center;
  background: #f7f7f2;
  border-radius: 6px;
  border: 1px solid #e0e0d8;
}

.high { font-weight: bold; color: #B3261E; }
.low { color: #5A5A5A; }
```

Code Listing 31.2: Weather Dashboard Styles

31.4 JavaScript Async Weather Engine

```
const form = document.querySelector("#weatherSearchForm");
const cityInput = document.querySelector("#cityInput");
const loadingEl = document.querySelector("#loadingSpinner");
const errorEl = document.querySelector("#errorBanner");
const contentEl = document.querySelector("#weatherContent");

async function fetchWeatherData(city) {
```

```
  // Free public Open-Meteo geocoding + weather API
  const geoUrl = `https://geocoding-api.open-meteo.com/v1/search?name=${
    ↪ encodeURIComponent(city)}&count=1`;

  const geoRes = await fetch(geoUrl);
  if (!geoRes.ok) throw new Error("Geocoding service unavailable");
  const geoData = await geoRes.json();

  if (!geoData.results || geoData.results.length === 0) {
    throw new Error(`City "${city}" not found. Please check spelling.`);
  }

  const { latitude, longitude, name, country } = geoData.results[0];

  const weatherUrl = `https://api.open-meteo.com/v1/forecast?latitude=${latitude}&
    ↪ longitude=${longitude}&current_weather=true&daily=temperature_2m_max,temperature_2
    ↪ m_min&timezone=auto`;
  const weatherRes = await fetch(weatherUrl);
  if (!weatherRes.ok) throw new Error("Weather forecast service failed");
  const weatherData = await weatherRes.json();

  return { name, country, weather: weatherData };
}

function renderWeather(data) {
  document.querySelector("#locationTitle").textContent = `${data.name}, ${data.country}`;
  document.querySelector("#currentTemp").textContent = `${Math.round(data.weather.
    ↪ current_weather.temperature)} deg C`;
  document.querySelector("#conditionText").textContent = `Wind: ${data.weather.
    ↪ current_weather.windspeed} km/h`;

  const dates = data.weather.daily.time;
  const maxTemps = data.weather.daily.temperature_2m_max;
  const minTemps = data.weather.daily.temperature_2m_min;

  const grid = document.querySelector("#forecastGrid");
  grid.innerHTML = dates.slice(1, 6).map((dateStr, idx) => {
    const dayName = new Intl.DateTimeFormat("en-US", { weekday: "short" }).format(new
    ↪ Date(dateStr));
    return `
      <div class="forecast-day-card">
        <h4>${dayName}</h4>
        <div class="high">${Math.round(maxTemps[idx + 1])} deg</div>
        <div class="low">${Math.round(minTemps[idx + 1])} deg</div>
      </div>
    `;
  }).join("");

  contentEl.hidden = false;
}

form.addEventListener("submit", async (e) => {
  e.preventDefault();
  const city = cityInput.value.trim();
  if (!city) return;

```

```
  loadingEl.hidden = false;
  errorEl.hidden = true;
  contentEl.hidden = true;

  try {
    const data = await fetchWeatherData(city);
    renderWeather(data);
  } catch (err) {
    errorEl.textContent = err.message;
    errorEl.hidden = false;
  } finally {
    loadingEl.hidden = true;
  }
});
```

Code Listing 31.3: Weather Planner JavaScript Engine

31.5 Testing Checklist and Extensions

- **Checklist**: Test invalid city queries (verify error banner), network offline mode, and API responses.
- **Extension 1**: Add a temperature unit toggle switch between Celsius and Fahrenheit.
- **Extension 2**: Save favorite cities in `localStorage` for quick one-click lookup.

32 MODULE

Project 7: Fedi Expense Tracker

The **Fedi Expense Tracker** is a complete personal financial budgeting application. It allows users to log income and expenses, calculates live net balance summaries, categorizes transactions, and persists all financial history in `localStorage`. This project solidifies your understanding of client-side CRUD operations and numeric reduction.

32.1 Project Overview and Requirements

LEARNING OBJECTIVES

- **Goal**: Build a full-featured budget manager with live balance summaries and persistence.
- **Concepts Applied**: Full CRUD in `localStorage`, Array `reduce()`, form validation, integer cents math.
- **Features**: Net balance display, positive/negative transaction styling, category tags, and delete actions.

32.2 HTML Layout Structure

```
<div class="expense-container">
  <h2>[FINANCE] Fedi Financial Tracker</h2>

  <div class="balance-summary-grid">
    <div class="summary-card net-balance">
      <span class="label">Net Balance</span>
      <h3 id="netBalance">$0.00</h3>
    </div>
    <div class="summary-card income">
      <span class="label">Total Income</span>
      <h4 id="totalIncome">+$0.00</h4>
    </div>
    <div class="summary-card expense">
      <span class="label">Total Expense</span>
      <h4 id="totalExpense">-$0.00</h4>
    </div>
  </div>

  <form id="transactionForm" class="transaction-form">
    <input type="text" id="descInput" placeholder="Description (e.g. Hosting, Coffee)"
    ↪ required />
```

```
    <input type="number" id="amountInput" placeholder="Amount ($)" step="0.01" required />
    <select id="typeSelect">
      <option value="expense">Expense</option>
      <option value="income">Income</option>
    </select>
    <button type="submit" class="btn btn-primary">Add Entry</button>
  </form>

  <h3>Transaction History</h3>
  <ul id="transactionList" class="transaction-list"></ul>
</div>
```

Code Listing 32.1: Expense Tracker HTML Structure

32.3 CSS Component Stylesheet

```
.expense-container {
  max-width: 580px;
  margin: 2rem auto;
  padding: 1.5rem;
  background: #ffffff;
  border-radius: 8px;
  border: 1px solid #d8d8ce;
}

.balance-summary-grid {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 1rem;
  margin: 1.5rem 0;
}

.summary-card {
  padding: 1rem;
  border-radius: 6px;
  background: #f7f7f2;
  text-align: center;
}

.summary-card.income h4 { color: #1E7A34; }
.summary-card.expense h4 { color: #B3261E; }

.tx-item {
  display: flex;
  justify-content: space-between;
  padding: 0.8rem;
  border-bottom: 1px solid #e0e0d8;
}

.tx-item.income { border-left: 4px solid #1E7A34; }
.tx-item.expense { border-left: 4px solid #B3261E; }
</end{jscode}
```

```
\section{JavaScript CRUD Logic}

\begin{jscode}[caption={Expense Tracker JavaScript Engine}]
const STORAGE_KEY = "fedi_expenses_v1";

function loadTransactions() {
  const raw = localStorage.getItem(STORAGE_KEY);
  return raw ? JSON.parse(raw) : [];
}

function saveTransactions(items) {
  localStorage.setItem(STORAGE_KEY, JSON.stringify(items));
}

let transactions = loadTransactions();

function renderTracker() {
  const list = document.querySelector("#transactionList");

  // Calculate Totals using reduce and integer cents
  const totals = transactions.reduce((acc, t) => {
    if (t.type === "income") acc.income += t.amountCents;
    else acc.expense += t.amountCents;
    return acc;
  }, { income: 0, expense: 0 });

  const netCents = totals.income - totals.expense;

  document.querySelector("#netBalance").textContent = `$${(netCents / 100).toFixed(2)}`;
  document.querySelector("#totalIncome").textContent = `+$${(totals.income / 100).toFixed
    ↪ (2)}`;
  document.querySelector("#totalExpense").textContent = `-$${(totals.expense / 100).
    ↪ toFixed(2)}`;

  list.innerHTML = transactions.map(t => `
    <li class="tx-item ${t.type}">
      <span class="tx-desc">${t.desc}</span>
      <span class="tx-amount">${t.type === "income" ? "+" : "-"}$${(t.amountCents / 100).
    ↪ toFixed(2)}</span>
      <button class="btn-delete" data-id="${t.id}">[Del]</button>
    </li>
  `).join("");
}

// Add Transaction
document.querySelector("#transactionForm").addEventListener("submit", (e) => {
  e.preventDefault();
  const desc = document.querySelector("#descInput").value.trim();
  const amountDollars = parseFloat(document.querySelector("#amountInput").value);
  const type = document.querySelector("#typeSelect").value;

  if (!desc || isNaN(amountDollars) || amountDollars <= 0) return;

  const newTx = {
    id: "TX-" + Date.now(),
```

```
    desc,
    amountCents: Math.round(amountDollars * 100),
    type
  };

  transactions.unshift(newTx);
  saveTransactions(transactions);
  renderTracker();
  e.target.reset();
});

// Delete Transaction via Event Delegation
document.querySelector("#transactionList").addEventListener("click", (e) => {
  if (e.target.matches(".btn-delete")) {
    const id = e.target.dataset.id;
    transactions = transactions.filter(t => t.id !== id);
    saveTransactions(transactions);
    renderTracker();
  }
});

// Initial Render
renderTracker();
```

Code Listing 32.2: Expense Tracker CSS Styles

32.4 Testing Checklist and Extensions

- **Checklist**: Verify that decimal calculations avoid float imprecision by verifying integer cent sums.
- **Extension 1**: Add CSV export functionality allowing users to download their financial records.
- **Extension 2**: Add a Canvas chart breakdown of expenses by category.

33 MODULE

Project 8: Fedi Password Generator

The **Fedi Password Generator** is a cybersecurity tool that creates cryptographically secure passwords. It provides character set toggles (uppercase, lowercase, numbers, symbols), a length slider, a live entropy-based password strength visualizer, and a one-click copy-to-clipboard button using the modern **Async Clipboard API**.

33.1 Project Overview and Requirements

LEARNING OBJECTIVES

- **Goal**: Generate cryptographically secure passwords and calculate theoretical entropy.
- **Concepts Applied**: Web Crypto API (`crypto.getRandomValues()`), regex matching, Clipboard API.
- **Features**: Configurable character sets, live entropy calculation, strength color meter, and copy feedback.

33.2 HTML Generator Layout

```
<div class="pwd-container">
  <h2>[SECURITY] Fedi Cryptographic Key Generator</h2>

  <div class="display-box">
    <input type="text" id="passwordOutput" readonly />
    <button id="copyBtn" class="btn btn-secondary">[COPY] Copy</button>
  </div>

  <div class="strength-meter">
    <div id="strengthBar" class="strength-fill"></div>
    <span id="strengthLabel">Strength: Moderate</span>
  </div>

  <div class="settings-box">
    <div class="setting-row">
      <label>Length: <span id="lengthVal">16</span></label>
      <input type="range" id="lengthSlider" min="8" max="64" value="16" />
    </div>
    <div class="checkbox-grid">
      <label><input type="checkbox" id="includeUpper" checked /> Uppercase (A-Z)</label>
      <label><input type="checkbox" id="includeLower" checked /> Lowercase (a-z)</label>
```

```
      <label><input type="checkbox" id="includeNumbers" checked /> Numbers (0-9)</label>
      <label><input type="checkbox" id="includeSymbols" checked /> Symbols (!@#$%)</label>
    </div>
  </div>

  <button id="generateBtn" class="btn btn-primary btn-block">Generate Secure Password</button>
</div>
```

Code Listing 33.1: Password Generator HTML Layout

33.3 CSS Security Console Stylesheet

```
.pwd-container {
  max-width: 520px;
  margin: 2rem auto;
  padding: 1.5rem;
  background: #ffffff;
  border: 1px solid #d8d8ce;
  border-radius: 8px;
}

.display-box {
  display: flex;
  gap: 0.5rem;
  margin: 1.2rem 0;
}

.strength-meter {
  height: 8px;
  background: #e0e0d8;
  border-radius: 4px;
  margin-bottom: 1.2rem;
  overflow: hidden;
}

.strength-fill {
  height: 100%;
  width: 50%;
  transition: width 0.3s ease, background-color 0.3s ease;
}

.checkbox-grid {
  display: grid;
  grid-template-columns: 1fr 1fr;
  gap: 0.8rem;
  margin: 1rem 0;
}
```

Code Listing 33.2: Password Generator Styles

33.4 JavaScript Cryptographic Engine

```
const CHAR_SETS = {
  upper: "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
  lower: "abcdefghijklmnopqrstuvwxyz",
  numbers: "0123456789",
  symbols: "!@#$%^&*()_+~`|}{[]:;?><,./-="
};

// Cryptographically secure random integer within range [0, max)
function getSecureRandomInt(max) {
  const array = new Uint32Array(1);
  crypto.getRandomValues(array);
  return array[0] % max;
}

function generatePassword() {
  const length = parseInt(document.querySelector("#lengthSlider").value);
  const useUpper = document.querySelector("#includeUpper").checked;
  const useLower = document.querySelector("#includeLower").checked;
  const useNumbers = document.querySelector("#includeNumbers").checked;
  const useSymbols = document.querySelector("#includeSymbols").checked;

  let pool = "";
  if (useUpper) pool += CHAR_SETS.upper;
  if (useLower) pool += CHAR_SETS.lower;
  if (useNumbers) pool += CHAR_SETS.numbers;
  if (useSymbols) pool += CHAR_SETS.symbols;

  if (pool.length === 0) {
    alert("Please select at least one character type!");
    return;
  }

  let password = "";
  for (let i = 0; i < length; i++) {
    const randIndex = getSecureRandomInt(pool.length);
    password += pool[randIndex];
  }

  document.querySelector("#passwordOutput").value = password;
  evaluateStrength(password, pool.length);
}

function evaluateStrength(password, poolSize) {
  // Information Entropy: E = L * log2(R)
  const entropy = password.length * Math.log2(poolSize);
  const bar = document.querySelector("#strengthBar");
  const label = document.querySelector("#strengthLabel");

  if (entropy < 40) {
    bar.style.width = "25%";
    bar.style.backgroundColor = "#B3261E";
    label.textContent = `Strength: Weak (${Math.round(entropy)} bits)`;
  } else if (entropy < 70) {
```

```
    bar.style.width = "60%";
    bar.style.backgroundColor = "#C9A227";
    label.textContent = `Strength: Moderate (${Math.round(entropy)} bits)`;
  } else {
    bar.style.width = "100%";
    bar.style.backgroundColor = "#1E7A34";
    label.textContent = `Strength: Very Strong (${Math.round(entropy)} bits)`;
  }
}

// Copy to Clipboard with Modern API
document.querySelector("#copyBtn").addEventListener("click", async () => {
  const pwd = document.querySelector("#passwordOutput").value;
  if (!pwd) return;

  await navigator.clipboard.writeText(pwd);
  const copyBtn = document.querySelector("#copyBtn");
  copyBtn.textContent = "[OK] Copied!";
  setTimeout(() => { copyBtn.textContent = "[COPY] Copy"; }, 2000);
});

document.querySelector("#generateBtn").addEventListener("click", generatePassword);
document.querySelector("#lengthSlider").addEventListener("input", (e) => {
  document.querySelector("#lengthVal").textContent = e.target.value;
  generatePassword();
});

// Initial Generation
generatePassword();
```

Code Listing 33.3: Password Generator Logic

33.5 Testing Checklist and Extensions

- **Checklist**: Verify that generated passwords always contain chosen characters; test Clipboard copy.
- **Extension 1**: Add a passphrase generator that combines random dictionary words.
- **Extension 2**: Check generated passwords against common leaked password hash databases.

34 MODULE

Project 9: Fedi Movie Search

The **Fedi Movie Search** application is a media discovery interface. It queries a remote film database API as the user types, implementing **debouncing** to prevent flooding the server with network requests, rendering responsive movie posters, and providing a bookmarking system to save favorite movies to a personal watchlist.

34.1 Project Overview and Requirements

LEARNING OBJECTIVES

- **Goal**: Build a debounced search engine with modal detail cards and a saved watchlist.
- **Concepts Applied**: Debouncing with closures and timers, REST API integration, modal dialogs, and `localStorage`.
- **Features**: 300ms input debounce, loading states, favorite toggle button, and poster grid.

34.2 HTML Layout Structure

```
<div class="movie-app-container">
  <header class="app-header">
    <h2>[CINEMA] Fedi Cinema Explorer</h2>
    <div class="search-box">
      <input type="text" id="movieSearchInput" placeholder="Search movies (e.g. Inception,
    ↪  Matrix, Interstellar)..." />
      <span id="searchStatus" class="status-indicator"></span>
    </div>
  </header>

  <div id="movieGrid" class="movie-grid"></div>
</div>
```

Code Listing 34.1: Movie Search HTML Layout

34.3 CSS Cinema Grid Stylesheet

```
.movie-app-container {
  max-width: 960px;
```

```
  margin: 2rem auto;
  padding: 1rem;
}

.search-box {
  margin: 1.5rem 0;
  position: relative;
}

.movie-grid {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
  gap: 1.5rem;
}

.movie-card {
  background: #ffffff;
  border-radius: 8px;
  overflow: hidden;
  border: 1px solid #d8d8ce;
  transition: transform 0.2s ease;
}

.movie-card:hover {
  transform: translateY(-4px);
}

.movie-card img {
  width: 100%;
  height: 280px;
  object-fit: cover;
}

.movie-info {
  padding: 0.8rem;
}
```

Code Listing 34.2: Cinema Explorer Styles

34.4 JavaScript Debounce and Search Engine

```
// 1. Debounce Utility (Higher-Order Function with Closure)
function debounce(fn, delayMs = 300) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {
      fn.apply(this, args);
    }, delayMs);
  };
}

// 2. Fetch Movie Data from Public Search Endpoint
```

```
async function searchMovies(query) {
  if (!query) {
    document.querySelector("#movieGrid").innerHTML = "";
    return;
  }

  const status = document.querySelector("#searchStatus");
  status.textContent = "Searching...";

  try {
    const url = `https://www.omdbapi.com/?s=${encodeURIComponent(query)}&apikey=trilogy`;
    const res = await fetch(url);
    const data = await res.json();

    if (data.Response === "True") {
      renderMovies(data.Search);
      status.textContent = `Found ${data.totalResults} titles`;
    } else {
      document.querySelector("#movieGrid").innerHTML = `<p class="no-results">No movies found matching "${query}".</p>`;
      status.textContent = "";
    }
  } catch (err) {
    console.error("Movie search error:", err);
    status.textContent = "Network error";
  }
}

function renderMovies(movies) {
  const grid = document.querySelector("#movieGrid");
  grid.innerHTML = movies.map(m => {
    const poster = m.Poster !== "N/A" ? m.Poster : "assets/images/no-poster.png";
    return `
      <div class="movie-card">
        <img src="${poster}" alt="${m.Title}" loading="lazy" />
        <div class="movie-info">
          <h4>${m.Title}</h4>
          <span class="year">${m.Year}</span>
        </div>
      </div>
    `;
  }).join("");
}

// Attach Debounced Input Handler
const debouncedSearch = debounce((e) => searchMovies(e.target.value.trim()), 400);
document.querySelector("#movieSearchInput").addEventListener("input", debouncedSearch);
```

Code Listing 34.3: Movie Search Engine Logic

34.5 Testing Checklist and Extensions

- **Checklist**: Type quickly and verify in DevTools Network tab that only one HTTP request fires after you stop typing.

- **Extension 1**: Add an IMDB rating sort dropdown and release year range slider.
- **Extension 2**: Build an infinite scroll pagination handler that loads subsequent pages as the user scrolls.

35 MODULE

Project 10: Fedi Task Manager with Classes

The **Fedi Task Manager** is an enterprise-grade productivity application designed using full **Object-Oriented Programming (OOP)**. It organizes tasks across status lanes (To Do, In Progress, Done) with priority levels, due dates, and completion timestamps. This project puts into practice ES6 classes, private fields, encapsulation, inheritance, and event delegation.

35.1 Project Architecture and Class Design

LEARNING OBJECTIVES

- **Goal**: Build a robust, modular Kanban task board using clean OOP architecture.
- **Concepts Applied**: ES6 Classes, private fields (`#fields`), static factory methods, state serialization.
- **Features**: Column-based task progression, priority tagging, and automatic timestamp tracking.

35.2 HTML Kanban Board Structure

```
<div class="kanban-app">
  <header class="app-header">
    <h2>[KANBAN] Fedi Task Management Suite</h2>
    <form id="newTaskForm" class="new-task-bar">
      <input type="text" id="taskTitleInput" placeholder="New Task Summary..." required />
      <select id="taskPrioritySelect">
        <option value="low">Low Priority</option>
        <option value="medium" selected>Medium Priority</option>
        <option value="high">High Priority</option>
      </select>
      <button type="submit" class="btn btn-primary">[+] Add Task</button>
    </form>
  </header>

  <div id="kanbanBoard" class="kanban-columns-grid">
    <div class="column-lane">
      <h3>To Do</h3>
      <div id="colTodo" class="cards-list"></div>
    </div>
    <div class="column-lane">
```

```
      <h3>In Progress</h3>
      <div id="colInProgress" class="cards-list"></div>
    </div>
    <div class="column-lane">
      <h3>Completed</h3>
      <div id="colDone" class="cards-list"></div>
    </div>
  </div>
</div>
```

Code Listing 35.1: Task Manager HTML Structure

35.3 CSS Kanban Board Stylesheet

```
.kanban-columns-grid {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 1.5rem;
  margin-top: 1.5rem;
}

.column-lane {
  background: #f7f7f2;
  border: 1px solid #d8d8ce;
  border-radius: 8px;
  padding: 1rem;
}

.task-card {
  background: #ffffff;
  border-radius: 6px;
  padding: 0.8rem;
  margin-bottom: 0.8rem;
  border: 1px solid #e0e0d8;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.04);
}

.task-card.priority-high { border-left: 4px solid #B3261E; }
.task-card.priority-medium { border-left: 4px solid #C9A227; }
.task-card.priority-low { border-left: 4px solid #1E7A34; }
</end{jscode}

\section{The Task and TaskManager Classes}

\begin{jscode}[caption={Object-Oriented Task and TaskManager Classes}]
class Task {
  #id;
  #title;
  #priority;
  #status;
  #createdAt;

  constructor(title, priority = "medium", status = "todo") {
```

```
    this.#id = "TASK-" + Math.random().toString(36).substr(2, 9);
    this.#title = title;
    this.#priority = priority; // "low", "medium", "high"
    this.#status = status;     // "todo", "in-progress", "done"
    this.#createdAt = new Date();
  }

  // Getters
  get id() { return this.#id; }
  get title() { return this.#title; }
  get priority() { return this.#priority; }
  get status() { return this.#status; }

  // Setters and State Transitions
  advanceStatus() {
    if (this.#status === "todo") this.#status = "in-progress";
    else if (this.#status === "in-progress") this.#status = "done";
  }

  toJSON() {
    return {
      id: this.#id,
      title: this.#title,
      priority: this.#priority,
      status: this.#status,
      createdAt: this.#createdAt.toISOString()
    };
  }

  static fromJSON(data) {
    const task = new Task(data.title, data.priority, data.status);
    return task;
  }
}

class TaskManagerApp {
  #tasks = [];
  #storageKey = "fedi_kanban_tasks";

  constructor() {
    this.#load();
    this.#bindEvents();
    this.render();
  }

  addTask(title, priority) {
    const task = new Task(title, priority);
    this.#tasks.push(task);
    this.#save();
    this.render();
  }

  advanceTask(taskId) {
    const task = this.#tasks.find(t => t.id === taskId);
    if (task) {
      task.advanceStatus();
```

```
      this.#save();
      this.render();
    }
  }

  deleteTask(taskId) {
    this.#tasks = this.#tasks.filter(t => t.id !== taskId);
    this.#save();
    this.render();
  }

  #save() {
    const serialized = this.#tasks.map(t => t.toJSON());
    localStorage.setItem(this.#storageKey, JSON.stringify(serialized));
  }

  #load() {
    const raw = localStorage.getItem(this.#storageKey);
    if (raw) {
      const data = JSON.parse(raw);
      this.#tasks = data.map(d => Task.fromJSON(d));
    }
  }

  #bindEvents() {
    document.querySelector("#newTaskForm").addEventListener("submit", (e) => {
      e.preventDefault();
      const title = document.querySelector("#taskTitleInput").value.trim();
      const priority = document.querySelector("#taskPrioritySelect").value;
      if (title) {
        this.addTask(title, priority);
        e.target.reset();
      }
    });

    document.querySelector("#kanbanBoard").addEventListener("click", (e) => {
      const card = e.target.closest(".task-card");
      if (!card) return;
      const taskId = card.dataset.id;

      if (e.target.matches(".btn-advance")) {
        this.advanceTask(taskId);
      } else if (e.target.matches(".btn-delete")) {
        this.deleteTask(taskId);
      }
    });
  }

  render() {
    const cols = {
      todo: document.querySelector("#colTodo"),
      "in-progress": document.querySelector("#colInProgress"),
      done: document.querySelector("#colDone")
    };

    Object.values(cols).forEach(col => { col.innerHTML = ""; });
```

```
152
153     this.#tasks.forEach(task => {
154       const card = document.createElement("div");
155       card.className = `task-card priority-${task.priority}`;
156       card.dataset.id = task.id;
157       card.innerHTML = `
158         <div class="card-header">
159           <span class="badge badge-${task.priority}">${task.priority}</span>
160           <button class="btn-delete">[x]</button>
161         </div>
162         <p class="task-title">${task.title}</p>
163         ${task.status !== "done" ? '<button class="btn btn-advance">Advance [->]</button
    ↪ >' : '<span class="completed-check">[DONE]</span>'}
164       `;
165       cols[task.status].append(card);
166     });
167   }
168 }
169
170 // Launch application
171 const app = new TaskManagerApp();
```

Code Listing 35.2: Kanban Columns CSS

35.4 Testing Checklist and Extensions

- **Checklist**: Verify that private fields cannot be manipulated directly in console; test status advancement.
- **Extension 1**: Add HTML5 Drag and Drop API support so users can drag task cards between Kanban columns.
- **Extension 2**: Add an overdue warning if a task with high priority remains in "todo" for more than 48 hours.

36 MODULE

Project 11: Nth Fibonacci Number Visualizer

The **Nth Fibonacci Visualizer** is an algorithmic laboratory application that directly compares three computation strategies: **Naive Recursion ($O(2^n)$)**, **Top-Down Memoization ($O(n)$)**, and **Bottom-Up Tabulation ($O(n)$)**. It visualizes the recursive execution call tree in real time, measures microsecond execution durations, and counts total function invocations, making abstract computational complexity visually concrete.

36.1 Project Architecture and Complexity Laboratory

LEARNING OBJECTIVES

- **Goal**: Build an algorithmic visualizer that dynamically animates recursive call trees and benchmarks DP.
- **Concepts Applied**: Recursion instrumentation, Dynamic Programming, DOM rendering, `performance.now()`.
- **Features**: Execution step counter, call stack depth meter, microsecond benchmark chart, and visual tree animator.

36.2 HTML Layout Structure

```
<div class="visualizer-container">
  <h2>[PERF] Fibonacci Computational Complexity Lab</h2>

  <div class="control-panel">
    <label for="nInput">Compute Nth Fibonacci Number (N = <span id="nVal">6</span>):</
    ↪ label>
    <input type="range" id="nInput" min="1" max="40" value="6" />

    <div class="strategy-radios">
      <label><input type="radio" name="strategy" value="naive" checked /> Naive Recursion
    ↪  O(2^n)</label>
      <label><input type="radio" name="strategy" value="memo" /> Memoized DP O(n)</label>
      <label><input type="radio" name="strategy" value="tab" /> Tabulation O(n)</label>
    </div>

    <button id="runBtn" class="btn btn-primary">Run Simulation</button>
```

```
  </div>

  <div class="metrics-grid">
    <div class="metric-card"><span>Result:</span><h3 id="resultVal">-</h3></div>
    <div class="metric-card"><span>Function Calls:</span><h3 id="callsVal">-</h3></div>
    <div class="metric-card"><span>Execution Time:</span><h3 id="timeVal">-</h3></div>
  </div>

  <div id="treeContainer" class="tree-canvas-box"></div>
</div>
```

Code Listing 36.1: Fibonacci Visualizer HTML Markup

36.3 CSS Laboratory Stylesheet

```
.visualizer-container {
  max-width: 680px;
  margin: 2rem auto;
  padding: 1.5rem;
  background: #ffffff;
  border-radius: 8px;
  border: 1px solid #d8d8ce;
}

.metrics-grid {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 1rem;
  margin: 1.5rem 0;
}

.metric-card {
  padding: 1rem;
  background: #f7f7f2;
  border-radius: 6px;
  text-align: center;
}

.metric-card h3 {
  color: #C9A227;
  font-size: 1.5rem;
  margin-top: 0.3rem;
}
</end{jscode}

\section{JavaScript Visualizer Engine}

\begin{jscode}[caption={Fibonacci Complexity Engine and Tree Generator}]
let callCount = 0;

// 1. Naive Recursive Fibonacci with tree node generation
function fibNaive(n, parentNode = null) {
  callCount++;
```

```
  const node = { name: `fib(${n})`, children: [] };
  if (parentNode) parentNode.children.push(node);

  if (n <= 1) {
    node.result = n;
    return { val: n, tree: node };
  }

  const left = fibNaive(n - 1, node);
  const right = fibNaive(n - 2, node);
  node.result = left.val + right.val;

  return { val: node.result, tree: node };
}

// 2. Memoized Fibonacci
function fibMemoized(n, memo = new Map()) {
  callCount++;
  if (memo.has(n)) return memo.get(n);
  if (n <= 1) return n;

  const res = fibMemoized(n - 1, memo) + fibMemoized(n - 2, memo);
  memo.set(n, res);
  return res;
}

// 3. Tabulated Fibonacci (Bottom-Up)
function fibTabulated(n) {
  callCount++;
  if (n <= 1) return n;
  let prev2 = 0, prev1 = 1;
  for (let i = 2; i <= n; i++) {
    callCount++;
    const cur = prev1 + prev2;
    prev2 = prev1;
    prev1 = cur;
  }
  return prev1;
}

function runBenchmark() {
  const n = parseInt(document.querySelector("#nInput").value);
  const strategy = document.querySelector('input[name="strategy"]:checked').value;
  callCount = 0;

  if (strategy === "naive" && n > 35) {
    alert("Warning: N > 35 with naive recursion may freeze the browser due to O(2^n)
    ↪ complexity!");
  }

  const start = performance.now();
  let result;

  if (strategy === "naive") {
    const { val } = fibNaive(n);
    result = val;
```

```
  } else if (strategy === "memo") {
    result = fibMemoized(n);
  } else {
    result = fibTabulated(n);
  }

  const end = performance.now();
  const duration = (end - start).toFixed(3);

  document.querySelector("#resultVal").textContent = result.toLocaleString();
  document.querySelector("#callsVal").textContent = callCount.toLocaleString();
  document.querySelector("#timeVal").textContent = `${duration} ms`;
}

document.querySelector("#runBtn").addEventListener("click", runBenchmark);
document.querySelector("#nInput").addEventListener("input", (e) => {
  document.querySelector("#nVal").textContent = e.target.value;
});
```

Code Listing 36.2: Fibonacci Complexity Lab Styles

36.4 Testing Checklist and Extensions

- **Checklist**: Compare $N = 30$ across all 3 strategies: observe naive takes $> 2,000,000$ calls, while memoization takes < 60 calls.
- **Extension 1**: Render an SVG tree diagram with green highlighted nodes for cached subproblem hits.
- **Extension 2**: Support BigInt values to compute $N = 1000$ without integer precision overflow.

37 MODULE

Project 12: Fedi Pathfinder

Welcome to the ultimate Capstone Project of the book: the **Fedi Pathfinder**. This project integrates everything you have learned across all 25 modules: HTML5/CSS grid layouts, asynchronous delays, Graph representations, Breadth-First Search (BFS), Depth-First Search (DFS), heuristics, and interactive animation. You will build a visual pathfinding sandbox where users can draw obstacle mazes on a 2D grid and watch search algorithms explore cells in real time to locate the optimal route.

37.1 Project Overview and Requirements

LEARNING OBJECTIVES

- **Goal**: Build an interactive 2D grid pathfinding visualizer with animated exploration waves.
- **Concepts Applied**: Graph modeling of 2D grids, Queue-based BFS, Stack-based DFS, async animations, mouse drag events.
- **Features**: Wall painting, moveable Start and Target nodes, speed slider, step counters, and yellow shortest-path trail.

37.2 HTML Grid and Controls Layout

```
<div class="pathfinder-app">
  <header class="app-toolbar">
    <h2>[MAP] Fedi Graph Pathfinder Visualizer</h2>
    <div class="controls-group">
      <select id="algoSelect">
        <option value="bfs">Breadth-First Search (Guaranteed Shortest)</option>
        <option value="dfs">Depth-First Search (Deep Exploration)</option>
      </select>
      <button id="startVisualBtn" class="btn btn-primary">[>] Find Path</button>
      <button id="clearWallsBtn" class="btn">Clear Walls</button>
      <button id="resetGridBtn" class="btn">Reset All</button>
    </div>
  </header>

  <div class="legend-bar">
    <span>[START] Start (0,0)</span>
    <span>[TARGET] Target (19,29)</span>
    <span>[#] Wall / Obstacle</span>
    <span>[*] Explored Cell</span>
    <span>[PATH] Shortest Path</span>
```

```
  </div>

  <div id="gridBoard" class="grid-board"></div>
</div>
```

Code Listing 37.1: Pathfinder HTML Markup

37.3 CSS Pathfinding Grid Stylesheet

```
.pathfinder-app {
  max-width: 900px;
  margin: 1.5rem auto;
  padding: 1rem;
}

.grid-board {
  display: grid;
  gap: 1px;
  background: #d8d8ce;
  margin: 1.5rem auto;
  user-select: none;
  width: fit-content;
}

.grid-cell {
  background: #ffffff;
  transition: background-color 0.2s ease;
}

.grid-cell.cell-wall { background: #1A1A1A; }
.grid-cell.cell-start { background: #1E7A34; }
.grid-cell.cell-target { background: #B3261E; }
.grid-cell.cell-visited { background: #EAF1FB; }
.grid-cell.cell-path { background: #F7DF1E; }
</end{jscode}

\section{JavaScript Pathfinding Engine}

\begin{jscode}[caption={Fedi Pathfinder Graph Engine and Visualizer}]
const ROWS = 20;
const COLS = 30;
let grid = [];
let isMousePressed = false;
let startPos = { r: 5, c: 5 };
let targetPos = { r: 15, c: 25 };
let isRunning = false;

// 1. Initialize Grid Matrix
function initGrid() {
  const board = document.querySelector("#gridBoard");
  board.style.gridTemplateRows = `repeat(${ROWS}, 22px)`;
  board.style.gridTemplateColumns = `repeat(${COLS}, 22px)`;
  board.innerHTML = "";
```

```
  grid = [];

  for (let r = 0; r < ROWS; r++) {
    const row = [];
    for (let c = 0; c < COLS; c++) {
      const cellEl = document.createElement("div");
      cellEl.className = "grid-cell";
      cellEl.dataset.r = r;
      cellEl.dataset.c = c;
      board.append(cellEl);

      const cellData = {
        r, c,
        isWall: false,
        element: cellEl
      };
      row.push(cellData);
    }
    grid.push(row);
  }

  updateSpecialCells();
}

function updateSpecialCells() {
  grid[startPos.r][startPos.c].element.classList.add("cell-start");
  grid[targetPos.r][targetPos.c].element.classList.add("cell-target");
}

// 2. Mouse Painting for Obstacle Walls
document.querySelector("#gridBoard").addEventListener("mousedown", (e) => {
  if (isRunning) return;
  isMousePressed = true;
  toggleWall(e);
});

window.addEventListener("mouseup", () => { isMousePressed = false; });

document.querySelector("#gridBoard").addEventListener("mouseover", (e) => {
  if (isMousePressed && !isRunning) toggleWall(e);
});

function toggleWall(e) {
  const el = e.target.closest(".grid-cell");
  if (!el) return;
  const r = parseInt(el.dataset.r);
  const c = parseInt(el.dataset.c);
  if ((r === startPos.r && c === startPos.c) || (r === targetPos.r && c === targetPos.c))
    ↪ return;

  const cell = grid[r][c];
  cell.isWall = !cell.isWall;
  cell.element.classList.toggle("cell-wall", cell.isWall);
}

// 3. Helper: Get Valid Orthogonal Neighbors
```

```
function getNeighbors(r, c) {
  const neighbors = [];
  const dirs = [[-1, 0], [1, 0], [0, -1], [0, 1]]; // Up, Down, Left, Right

  for (const [dr, dc] of dirs) {
    const nr = r + dr;
    const nc = c + dc;
    if (nr >= 0 && nr < ROWS && nc >= 0 && nc < COLS) {
      if (!grid[nr][nc].isWall) {
        neighbors.push(grid[nr][nc]);
      }
    }
  }
  return neighbors;
}

// 4. Animated BFS Algorithm Implementation
async function visualizeBFS() {
  isRunning = true;
  const queue = [grid[startPos.r][startPos.c]];
  const visited = new Set([`${startPos.r},${startPos.c}`]);
  const parentMap = new Map();
  let found = false;

  while (queue.length > 0) {
    const current = queue.shift();

    if (current.r === targetPos.r && current.c === targetPos.c) {
      found = true;
      break;
    }

    // Don't recolor the start cell
    if (!(current.r === startPos.r && current.c === startPos.c)) {
      current.element.classList.add("cell-visited");
    }

    // Visual animation pause (10ms)
    await new Promise(res => setTimeout(res, 10));

    const neighbors = getNeighbors(current.r, current.c);
    for (const neighbor of neighbors) {
      const key = `${neighbor.r},${neighbor.c}`;
      if (!visited.has(key)) {
        visited.add(key);
        parentMap.set(key, current);
        queue.push(neighbor);
      }
    }
  }

  if (found) {
    await traceShortestPath(parentMap);
  } else {
    alert("No path exists to target!");
  }
```

```
  isRunning = false;
}

// 5. Shortest Path Reconstruction
async function traceShortestPath(parentMap) {
  let curr = grid[targetPos.r][targetPos.c];
  const path = [];

  while (curr) {
    path.push(curr);
    const key = `${curr.r},${curr.c}`;
    curr = parentMap.get(key);
  }

  path.reverse();
  for (const cell of path) {
    if (!(cell.r === startPos.r && cell.c === startPos.c) && !(cell.r === targetPos.r &&
    ↪ cell.c === targetPos.c)) {
      cell.element.classList.add("cell-path");
      await new Promise(res => setTimeout(res, 25));
    }
  }
}

// Event Bindings
document.querySelector("#startVisualBtn").addEventListener("click", () => {
  if (!isRunning) visualizeBFS();
});

document.querySelector("#clearWallsBtn").addEventListener("click", () => {
  for (let r = 0; r < ROWS; r++) {
    for (let c = 0; c < COLS; c++) {
      grid[r][c].isWall = false;
      grid[r][c].element.className = "grid-cell";
    }
  }
  updateSpecialCells();
});

initGrid();
```

Code Listing 37.2: Pathfinder Matrix CSS

37.4 Testing, Diagnostics, and Capstone Mastery

- **Checklist**: Draw a complete wall partition around the target; verify that BFS reports unreachable after exploring all accessible nodes.
- **Extension 1**: Implement the **A* Search Algorithm** using the Manhattan Distance heuristic ($|r_1 - r_2| + |c_1 - c_2|$) for ultra-fast targeted routing.
- **Extension 2**: Add a randomized Recursive Backtracking Maze Generator button to automatically carve complex labyrinths.

PART VI

JavaScript Step S — Master Review

38 MODULE

Master Review and Progression Map

You have journeyed across the entire landscape of modern JavaScript. What began with declaring variables and logging strings has culminated in building pathfinding graph engines, architecting object hierarchies, and optimizing dynamic programming algorithms. This section synthesizes the core learnings of each tier into a bird's-eye architectural view.

38.1 The Four Tiers of JavaScript Mastery

38.1.1 Tier 1: Beginner — The Language Fundamentals

- **Variables & Types**: Primitive values stored by value on the stack (`string`, `number`, `boolean`, `null`, `undefined`, `symbol`, `bigint`).
- **Declarations**: Always default to `const`; use `let` when reassignment is strictly required; never use legacy `var`.
- **Functions**: Declarations are hoisted; arrow functions provide lexical `this` and concise expressions; closures remember lexical parent environments.
- **Collections**: Arrays model ordered dynamic lists; objects model key-value dictionaries stored by reference in heap memory.

38.1.2 Tier 2: Intermediate — The Modern Web Platform

- **Declarative Pipelines**: Transform collections cleanly with Higher-Order Functions (`map`, `filter`, `reduce`, `sort`).
- **DOM Manipulation**: The browser creates a live node tree; query with `querySelector`, update with `textContent`, and handle events with high-performance event delegation.
- **Accessibility (a11y)**: Semantic markup and ARIA states guarantee usability for assistive technologies.
- **Validation & Storage**: Manage forms with the Constraint Validation API and persist client state with `localStorage`.

38.1.3 Tier 3: Advanced — Software Engineering & Algorithms

- **Object-Oriented Design**: Encapsulate private state with `#privateFields`; reuse behavior with `extends` and `super`; understand prototype chains.
- **Recursion**: Solve problems by decomposing into base cases and recursive steps; track call stack winding and unwinding.
- **Data Structures**: Implement Stacks (LIFO), Queues (FIFO), Linked Lists, Binary Search Trees, and Hash Tables.
- **Algorithmic Complexity**: Measure performance with Big O Notation; write $O(\log n)$ binary searches and $O(n \log n)$ merge sorts.

38.1.4 Tier 4: Master — Concurrency, Architecture & Scale

- **Graphs & Networks**: Model non-linear relations with Adjacency Lists; traverse with Breadth-First Search (BFS) and Depth-First Search (DFS).
- **Dynamic Programming**: Optimize exponential algorithms to polynomial time using Top-Down Memoization and Bottom-Up Tabulation.
- **Functional Programming**: Write pure, referentially transparent functions; build point-free pipelines with `pipe()` and currying.
- **Asynchronous JavaScript**: The Event Loop coordinates the Call Stack, Web APIs, Microtask Queue (Promises), and Macrotask Queue (`setTimeout`).

39 MODULE

Syntax and Methods Cheat Sheets

This section provides quick-reference lookup tables for the syntax, methods, and APIs introduced throughout the book. Keep these pages bookmarked as you build real-world applications.

39.1 Variables and Operators

Syntax	Description
`const x = val;`	Block-scoped constant (cannot be reassigned)
`let x = val;`	Block-scoped variable (reassignable)
`a === b`	Strict equality (compares type and value)
`a !== b`	Strict inequality
`a && b`	Logical AND (short-circuits on falsy)
`a || b`	Logical OR (short-circuits on truthy)
`a ?? b`	Nullish coalescing (falls back only on `null`/`undefined`)
`condition ? x : y`	Ternary conditional operator

39.2 Essential Array Methods

Method	Mutates?	Description
`push(...items)`	Yes	Adds to end; returns new length
`pop()`	Yes	Removes and returns last item
`unshift(...items)`	Yes	Adds to start; returns new length
`shift()`	Yes	Removes and returns first item
`slice(start, end)`	No	Returns shallow copy of sub-array
`splice(i, count, ...items)`	Yes	Inserts, removes, or replaces in-place
`map(fn)`	No	Transforms each element $1 \to 1$
`filter(fn)`	No	Returns subset of elements satisfying condition
`reduce(fn, init)`	No	Accumulates array into a single value
`find(fn)`	No	Returns first matching element (or `undefined`)
`findIndex(fn)`	No	Returns index of first match (or `-1`)
`some(fn)`	No	Returns `true` if at least one item matches
`every(fn)`	No	Returns `true` if all items match
`sort((a, b) => a - b)`	Yes	Sorts numerically ascending in place
`includes(val)`	No	Checks if value is present

39.3 Essential String Methods

Method	Description
`str.toUpperCase()`	Returns uppercase string
`str.toLowerCase()`	Returns lowercase string
`str.trim()`	Removes leading and trailing whitespace
`str.includes(sub)`	Returns `true` if substring exists
`str.indexOf(sub)`	Returns first index of substring (or `-1`)
`str.slice(start, end)`	Extracts substring from start up to end
`str.replace(a, b)`	Replaces first match with replacement
`str.replaceAll(a, b)`	Replaces all occurrences
`str.split(delim)`	Splits string into array of substrings
`` `Hello ${name}` ``	Template literal with embedded expression

39.4 DOM Manipulation and Events

DOM Method	Description
`document.querySelector(sel)`	Returns first matching element (or `null`)
`document.querySelectorAll(sel)`	Returns static `NodeList` of all matches
`el.textContent = "text"`	Safely sets text content without HTML parsing
`el.classList.add/remove/toggle()`	Modifies CSS classes
`el.setAttribute(attr, val)`	Sets HTML attribute
`el.append(...nodes)`	Appends elements or strings as child nodes
`el.remove()`	Removes element from DOM tree
`el.addEventListener(ev, fn)`	Attaches event listener
`e.preventDefault()`	Cancels default browser action (e.g. form submission)
`e.stopPropagation()`	Stops event from bubbling up DOM tree

39.5 Asynchronous JavaScript

Async Pattern	Description
`async function f() { ... }`	Declares function that returns a Promise
`const data = await promise;`	Pauses async execution until Promise resolves
`fetch(url, options)`	Native HTTP network request (returns Promise)
`Promise.all([p1, p2])`	Runs promises in parallel; rejects on first failure
`Promise.allSettled([p1, p2])`	Runs in parallel; returns status for all promises
`Promise.race([p1, p2])`	Settles as soon as the first promise settles

40 MODULE

Technical Interview Questions

This section contains top JavaScript technical interview questions drawn from real hiring processes at leading software engineering organizations. Use these questions to test your depth of understanding and practice communicating technical concepts with precision.

40.1 Core Language and Runtime

Q1. Explain the JavaScript Event Loop and the difference between Microtasks and Macrotasks.
Answer: JavaScript is single-threaded with one Call Stack. Asynchronous callbacks are queued in two places: (1) The **Microtask Queue** (Promises, `queueMicrotask`, `MutationObserver`), and (2) The **Task / Macrotask Queue** (`setTimeout`, `setInterval`, I/O, UI rendering). Whenever the Call Stack empties, the Event Loop drains the entire Microtask Queue before processing the next single Macrotask.

Q2. What is a closure and what are its practical use cases?
Answer: A closure is the combination of a function bundled with references to its surrounding lexical environment. It allows an inner function to retain access to outer variables even after the outer function has returned. Common use cases include: data encapsulation/privacy (private variables), function factories, currying, event handler state, and memoization.

Q3. What is the Temporal Dead Zone (TDZ)?
Answer: The TDZ is the period between the entering of a block scope and the actual line where a `let` or `const` variable is declared. Accessing the variable in the TDZ throws a `ReferenceError`, unlike `var` which is hoisted and returns `undefined`.

Q4. How does prototypal inheritance work in JavaScript?
Answer: Objects in JavaScript have an internal reference to a prototype object (`[[Prototype]]`). When accessing a property on an object, the engine searches the object itself; if not found, it traverses up the **prototype chain** until the property is located or `null` is reached. ES6 classes compile down to prototype delegation.

Q5. Why does `typeof null === "object"` in JavaScript?
Answer: This is a historical bug from the first version of JavaScript in 1995. Values were represented with a type tag in the lower bits of their memory words; the object type tag was `000`, and `null` was represented as a NULL pointer (`0x00`), causing the runtime to incorrectly report it as an object.

40.2 DOM and Web APIs

Q6. What is Event Delegation and what problems does it solve?
Answer: Event Delegation is attaching a single event listener to a common ancestor instead of attaching individual listeners to many child nodes. It leverages **event bubbling**, drastically

reducing memory consumption and eliminating the need to bind listeners to dynamically created child elements.

Q7. Why should sensitive authentication tokens NOT be stored in `localStorage`?
Answer: `localStorage` is accessible to any JavaScript running on the page. In the event of a Cross-Site Scripting (XSS) attack, malicious scripts can read and exfiltrate tokens. `HttpOnly` cookies are preferred because browsers prevent client-side JavaScript from accessing them.

Q8. What is the difference between `debounce` and `throttle`?
Answer: **Debounce** delays execution until N milliseconds have elapsed since the *last* invocation (ideal for search autocomplete inputs). **Throttle** limits execution to at most once every N milliseconds regardless of how frequently events fire (ideal for scroll or resize listeners).

40.3 Algorithms and Data Structures

Q9. How do you solve the Two Sum problem in $O(n)$ time?
Answer: Use a Hash Map. As you iterate through the array, calculate the complement (`target - num`). If the complement exists in the Map, return the stored index and current index. Otherwise, store `{ num: index }`. This achieves $O(n)$ time with $O(n)$ auxiliary space.

Q10. When would you choose a Breadth-First Search (BFS) over a Depth-First Search (DFS)?
Answer: Choose BFS when searching for the shortest path or closest node in an unweighted graph, or when exploring nodes level by level. Choose DFS when solving mazes, exploring complete branches, checking for graph cycles, or performing topological sorts.

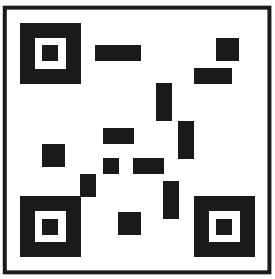
41 MODULE

Advanced Project Ideas for Master Practice

Theoretical mastery of JavaScript is only half the battle; true craftsmanship comes from building real-world software. This appendix compiles eight structured, challenging project ideas designed to push your skills to their absolute limits. These projects cover every tier of the book — from core fundamentals to advanced asynchronous event-driven architectures and algorithms.

To help you on your journey, the author has posted complete, production-grade solutions and code reviews for all of these projects on GitHub. You can scan the QR code below or visit the repository directly at:

https://github.com/fedixscar



Scan to Access Project Corrections

Figure 41.1: Scan the QR Code to Visit the GitHub Solutions Repository

41.1 Tier 1: Beginner Portfolio Additions

41.1.1 Project 1: The Personal Finance CLI Ledger

Focus: Scoping, loops, arrays, objects, and basic mathematical operations.

- **Concept**: A console-based financial manager that tracks incomes and expenses, categorizes transactions, and computes monthly tax brackets based on local regulations.
- **Key Requirements**:
  1. Store transactions as an array of objects, each containing an ID, amount, description, type (`"income"` or `"expense"`), and category.
  2. Build a console menu using `prompt()` to add, view, filter by category, and calculate balance.
  3. Implement a tax estimation routine that calculates marginal tax rates for FediUser's income.
- **Learning Outcome**: Solidifies object modeling, array searching, conditional control flow, and input casting.

41.1.2 Project 2: Fedi's Task Automator Library

Focus: Functions, lexical closures, parameters, callback structures.

- **Concept**: A utility module that registers scheduled tasks, runs them sequentially or conditionally, and exposes a stateful dashboard using closures.
- **Key Requirements**:
  1. Create a task registration function returning execution control interfaces.
  2. Enforce private registry storage using closures (no global variables allowed).
  3. Support callbacks to allow custom pre-execution checks and logging filters.
- **Learning Outcome**: Eliminates global namespace pollution and masters state encapsulation using closures.

41.2 Tier 2: Intermediate Dynamic Web Applications

41.2.1 Project 3: The Interactive Recipe Book with LocalStorage CRUD

Focus: DOM manipulation, event handling, JSON serialization, and state persistence.

- **Concept**: A web-based cook book where users add recipes, filter them dynamically by cooking time or dietary requirements, and persist their custom entries across page reloads.
- **Key Requirements**:
  1. Build an HTML interface with input forms for recipe name, ingredients, and steps.
  2. Save recipe arrays to `localStorage`; handle empty or corrupted keys with `try...catch` validation blocks.
  3. Use event delegation on list containers to handle dynamically added edit and delete buttons.
- **Learning Outcome**: Masters DOM traversal, form values extraction, and persistent local storage sync.

41.2.2 Project 4: The Accessible Markdown Editor

Focus: Accessibility, keyboard traps, live regions, regex parsing.

- **Concept**: A split-screen text editor that compiles Markdown syntax into HTML preview in real time, fully compliant with WCAG AA accessibility standards.
- **Key Requirements**:
  1. Use regular expressions to parse Markdown headers, links, and bold text into semantic HTML nodes.
  2. Focus trap key shortcuts: pressing `Ctrl + P` must shift focus between the editing area and the preview area instantly.
  3. Announce compiler status changes dynamically to screen readers using `aria-live="polite"` regions.
- **Learning Outcome**: Builds empathy and skill in accessible development, regex composition, and focus management.

41.3 Tier 3: Advanced Architecture and Algorithms

41.3.1 Project 5: Event-Driven E-Commerce Cart Engine

Focus: ES6 Classes, prototypal inheritance, custom event dispatcher.

- **Concept**: A backend cart manager class that handles item selection, currency conversions, discount promotions, and emits notifications to separate UI subscriber modules.
- **Key Requirements**:
  1. Write a base `EventEmitter` class to manage listener arrays, then extend it to create the `CartEngine`.
  2. Define private fields for items, quantities, and coupons. Provide public getters/setters for total estimations.
  3. Dispatch custom events (`cart:add`, `cart:checkout`, `cart:error`) with detail payloads containing item metadata.
- **Learning Outcome**: Deepens understanding of component decoupling, Class extensions, and custom event orchestration.

41.3.2 Project 6: Visual Call Stack Simulator

Focus: Recursion tracing, DOM tree generation, CSS transformations.

- **Concept**: An educational tool that allows users to write simple recursive functions (like factorial or Fibonacci) and visually inspect the Call Stack frames popping and pushing in real time.
- **Key Requirements**:
  1. Represent each execution context as a visual stack box block in the DOM.
  2. Implement delay steps to show step-by-step frame initialization.
  3. Highlight base cases and track variable environments dynamically in separate inspectors.
- **Learning Outcome**: Clarifies Call Stack lifecycles, execution memory, and dynamic UI rendering.

41.4 Tier 4: Master Concurrency and Optimization

41.4.1 Project 7: Asynchronous GitHub Dashboard with Retry/Timeout

Focus: Fetch API, AbortController, Promise combinators, rate limit caches.

- **Concept**: A developer profile dashboard that pulls repositories, stars, and user events from the GitHub REST API concurrently, handling API rate limits and network errors resiliently.
- **Key Requirements**:
  1. Fetch user profiles and repositories in parallel using `Promise.allSettled()` to prevent one failed query from breaking the whole page.
  2. Use `AbortController` to set a strict 3-second network timeout. If a request times out, retry it up to 3 times with exponential backoff.
  3. Implement a `Map`-based local cache storing queries for 5 minutes to avoid hitting rate limits.
- **Learning Outcome**: Masters professional API consumption, concurrency coordination, and error resilience.

41.4.2 Project 8: Dynamic Programming Grid Path Visualizer

Focus: Dynamic programming, BFS/DFS pathfinding, Matrix graph representations, canvas rendering.

- **Concept**: An interactive grid visualizer where users place obstacles and a start/end node, and the engine calculates and renders the optimal path using Dynamic Programming or Breadth-First Search (BFS).
- **Key Requirements**:
  1. Represent the grid as a 2D matrix array.
  2. Implement the Unique Grid Paths DP recurrence equation to compute the count of all paths.

3. Implement BFS to trace the shortest path, animating the node exploration queue state in real time.

- **Learning Outcome**: Synthesizes advanced non-linear data structures, memoization caching, and graph algorithms into a complex interactive visual app.

42 MODULE

Glossary of Terms

Accessibility (a11y) The practice of designing web applications to be usable by everyone, including individuals relying on assistive devices or keyboards.

Adjacency List A graph representation where each vertex stores a list or set of adjacent neighboring vertices.

Algorithm An unambiguous step-by-step procedure for performing a calculation or solving a computational problem.

Array An ordered, dynamic list collection indexed from 0 to `length - 1`.

Arrow Function A concise function syntax introduced in ES6 (`() => {}`) that lexically binds `this`.

Asynchronous Code that executes outside the primary synchronous execution thread without blocking execution.

Big O Notation A mathematical notation used to classify algorithms according to how their run time or space requirements grow as input size grows.

Binary Search An $O(\log n)$ search algorithm that repeatedly divides a sorted search interval in half.

Binary Search Tree (BST) A node-based binary tree data structure where left children are strictly less than parent, and right children are greater.

Block Scope Scope restricted to the enclosing curly braces `{ ... }` for variables declared with `let` or `const`.

Boolean A primitive data type that can have only two values: `true` or `false`.

Call Stack The data structure used by the JavaScript interpreter to keep track of function execution calls in LIFO order.

Callback A function passed as an argument to another function, intended to be executed at a later time.

Class An ES6 syntactic blueprint for creating objects with constructors, methods, private fields, and inheritance.

Closure A function bundled together with references to its surrounding lexical environment.

CRUD An acronym for the four basic persistent storage operations: Create, Read, Update, Delete.

Currying The technique of transforming a function of multiple arguments into a sequence of unary (single-argument) functions.

Debounce An optimization pattern that delays function invocation until a specific duration has passed since the last call.

Destructuring A syntax enabling the extraction of properties from objects or items from arrays into distinct variables.

Document Object Model (DOM) A tree representation of HTML documents created by browsers for programmatic manipulation.

Dynamic Programming An optimization method that breaks problems into overlapping subproblems, storing solved results to avoid recomputation.

Event Delegation Attaching an event listener to a parent container to manage events from all current and future child elements via event bubbling.

Event Loop The mechanism that constantly monitors the Call Stack and moves callbacks from queues to the stack when empty.

Higher-Order Function A function that accepts another function as an argument or returns a function.

Hoisting The JavaScript engine behavior of lifting variable and function declarations to the top of their enclosing scope during compilation.

Immutability The property of a value or data structure whose state cannot be modified after it is created.

JSON JavaScript Object Notation; a lightweight text-based data interchange format.

Lexical Scope Variable scope determined statically by the physical location of declarations in the source code.

Linked List A linear collection of data elements (nodes) where each node points to the next node via a reference.

Map An ES6 data structure representing an ordered collection of key-value pairs where keys can be of any data type.

Memoization An optimization technique that speeds up programs by storing the results of expensive function calls.

Microtask Queue A high-priority queue for Promise callbacks and async microtasks processed immediately when the Call Stack empties.

Object An unordered collection of key-value properties stored in heap memory and passed by reference.

Promise An object representing the eventual completion (or failure) of an asynchronous operation and its resulting value.

Pure Function A function whose return value depends solely on its arguments and produces no observable side effects.

Queue A First-In, First-Out (FIFO) data structure.

Recursion A programming technique where a function calls itself to solve smaller instances of the same problem.

Set An ES6 collection of unique values with no duplicates.

Stack A Last-In, First-Out (LIFO) data structure.

Comparison operator that checks both value and type without performing automatic type coercion.

Strict Equality (Tabulation A bottom-up dynamic programming approach that iteratively fills a table from base cases up.

Template Literal String literals enclosed in backticks (`` `...` ``) allowing embedded expressions (`${expr}`) and multi-line text.

WeakMap A Map collection holding weak references to object keys, allowing automatic garbage collection when no other references exist.

43 MODULE

Conclusion: The Journey Ahead

You have reached the final milestone of **JavaScript: From Beginner to Master**. Take a moment to appreciate how far you have traveled. You began by learning how memory reserves a simple variable box and outputs a string to the browser console. You have concluded by implementing graph traversal algorithms, optimizing dynamic programming states, orchestrating asynchronous concurrency, and building twelve interactive applications.

43.1 The Senior Engineer's Mindset

Technical mastery is not merely memorizing syntax or APIs. It is a disciplined way of approaching problems. As you transition into professional software development, carry these four core engineering principles with you:

1. **Write for Readability**: Code is read ten times more often than it is written. Choose clear names, avoid clever tricks, and structure functions to do exactly one thing well.
2. **Embrace Errors as Diagnostics**: When your program crashes, do not panic. Read the stack trace calmly from the top down. Errors are precise reports telling you exactly what the runtime expected versus what it received.
3. **Understand the Mechanics**: When you use an abstraction — whether a framework like React, a build tool like Vite, or an array method like `reduce` — strive to understand what it does under the hood. Understanding memory references, the prototype chain, and the Event Loop gives you superpowers when debugging complex systems.
4. **Build Genuinely**: The fastest way to grow is to build things you care about. Take the twelve capstone projects in this book, fork them, break them, redesign them, and deploy them to the live web for real users.

43.2 From Beginner to Master: The Self-Assessment Checklist

Review this checklist to verify your complete progression:

- ☐ I can explain the difference between `const`, `let`, and `var`, and I default to `const`.
- ☐ I understand how primitive values and object references are stored and manipulated in memory.
- ☐ I can explain how closures work and why they remember their lexical scope.
- ☐ I transform array data declaratively using `map`, `filter`, `reduce`, and method chaining.
- ☐ I can manipulate DOM nodes safely with `textContent` and use event delegation for high performance.
- ☐ I write accessible web components with semantic HTML and appropriate ARIA attributes.
- ☐ I use the browser debugger and breakpoints to diagnose bugs without relying solely on `console.log`.

- [ ] I can validate user input with regular expressions and the Constraint Validation API.
- [ ] I persist client application state across sessions with `localStorage` and JSON serialization.
- [ ] I can design modular class hierarchies with private fields (`#fields`) and inheritance.
- [ ] I can trace recursive functions and prevent stack overflow errors with clear base cases.
- [ ] I can implement Stacks, Queues, Linked Lists, and Binary Search Trees in JavaScript.
- [ ] I analyze the time and space complexity of algorithms using Big O Notation.
- [ ] I can traverse non-linear networks using Breadth-First Search (BFS) and Depth-First Search (DFS).
- [ ] I optimize exponential recursive algorithms using Top-Down Memoization and Bottom-Up Tabulation.
- [ ] I write pure, referentially transparent functions and compose them into functional pipelines.
- [ ] I understand how the Event Loop coordinates the Call Stack, Web APIs, Microtasks, and Macrotasks.
- [ ] I can fetch data from REST APIs using `async`/`await` and handle errors robustly with `try...catch`.

43.3 Where to Go Next

With the foundational and advanced principles of modern JavaScript firmly mastered, the entire JavaScript ecosystem is open to you:

- **TypeScript**: Add static type safety, interfaces, and compile-time tooling to large-scale JavaScript codebases.
- **Full-Stack Web (Node.js, Express, Next.js)**: Build scalable backend APIs, database connections, and server-rendered web applications.
- **Frontend Frameworks (React, Vue, Svelte)**: Build rich, reactive component-based user interfaces.
- **System & Mobile Development**: Build cross-platform mobile apps with React Native or desktop applications with Electron and Tauri.

“Some are born to obey. Some are cursed to see.”

Friedrich Wilhelm Nietzsche

Index

$_{p}roto_{,36}$

AbortController, 168
accessibility, 64
accessibility testing, 69
accessible forms, 68
adjacency list, 148
adjacency matrix, 148
algorithms
    searching, 138
    sorting, 138
AnalyserNode, 99
anti-patterns, 44
apply, 33
ARIA, 65
aria-expanded, 65
aria-label, 65
aria-live, 65, 67
array
    creation, 27
    indexing, 28
Array.from, 27
Array.isArray, 27
arrays
    transformations, 53
arrow function, 22
assignment operator, 3
assistive technology, 64
async-await, 167
asynchronous errors, 73
audio, 96
audio visualization, 99
AudioContext, 98
axe-core, 69

backreferences, 80
base case, 124
BFS, 149
Big O notation, 137
binary search, 138
binary search tree, 147
binary tree, 145
bind, 33
boolean, 12
bottom-up DP, 154
bracket notation, 33
breadth-first search, 149
break, 40
breakpoints, 75
BST, 133
    insert, 147
    search, 147
bubble sort, 138

call, 33
call stack
    errors, 72
    recursion, 125
callback hell, 166
callbacks, 52
cancellation, 168
catastrophic backtracking, 81
classes
    private fields, 118
    syntax, 117
classList, 58
climbing stairs, 155
closure, 24
closures
    functional pattern, 163
coin change problem, 156
color contrast, 68
comparison operators, 12
complexity analysis, 137
compose, 161
computed properties, 32
concatenation, 6
concurrency, 165
console methods, 75
const, 4
Constraint Validation API, 85
constructor, 117
continue, 40
cookies, 109
countdown timer, 94
cross-field validation, 87
CRUD operations, 112
currying, 161
custom audio player, 97
custom errors, 74

CustomEvent, 61

data structures
  binary search tree, 133
  linked list, 132
  queue, 131
  stack, 130
date arithmetic, 92
Date object, 90
debounce, 60
debugging
  beginner traps, 44
declarative programming, 162
default parameters, 23
depth-first search, 150
destructuring
  arrays, 30
detached DOM, 76
DevTools, 75
DFS, 150
DocumentFragment, 59
DOM, 57
  performance, 59
  selecting elements, 57
DOM tree, 57
dot notation, 33
dynamic programming, 153

edge, 148
Error subclassing, 74
errors
  types, 71
event bubbling, 59
event capturing, 59
event delegation, 60
event loop, 165
event propagation, 59
every, 54
exception propagation, 72
extends, 119

factorial, 125
falsy, 14
fast and slow pointers, 139
Fetch API, 167
Fibonacci, 125
FIFO, 131
filter, 53
finally block, 72
find, 54
findIndex, 54
first-class functions, 52
floating-point precision, 17
focus management, 66
form submission, 87
form validation, 84
FormData, 87
function composition, 161
function declaration, 20
function expression, 21
fundamentals review, 44

garbage collection, 106
generators, 41
getDate, 91
getFullYear, 91
getMonth, 91
getters, 35, 119
getUserMedia, 100
graphs, 148
grid paths, 156

hash function, 134
hash table, 134
heap snapshot, 76
holey arrays, 28
HTML5 validation, 84
HTMLCollection, 57
HTMLMediaElement, 96
HTTP methods, 167

immutability, 9, 160
  sorting, 54
imperative programming, 162
inheritance, 119
inline errors, 86
inorder, 146
input sanitization, 88
internationalization, 92
IntersectionObserver, 62
Intl.DateTimeFormat, 92
Intl.RelativeTimeFormat, 93

JSON.parse
  reviver, 110
JSON.stringify
  replacer, 110

key-value pairs, 103
keyboard navigation, 66

labeled statements, 40

labels, 68
lazy evaluation, 41
LCS, 157
let, 4
LIFO, 130
Lighthouse, 69
linked list
    doubly, 132
live regions, 67
localStorage, 109
    methods, 109
logical operators, 13
longest common subsequence, 157
loops
    do-while, 39
    for, 38
    for-in, 39
    for-of, 39
    while, 39

macrotasks, 165
Map, 103
    iteration, 105
    object keys, 104
    vs Object, 104
map, 53
match, 80
matchAll, 80
Math object, 16
media events, 97
MediaRecorder, 100
memoization, 154
memory
    classes, 121
    stack vs heap, 34
memory leaks, 76
merge sort, 138
method chaining, 55
methods, 33
microtasks, 165
mixins, 120
multidimensional array, 29
multiple inheritance, 120
mutability, 28
MutationObserver, 62

naming conventions, 5
new.target, 119
NodeList, 57
number, 15

Object.freeze, 36, 160
Object.preventExtensions, 36
Object.seal, 36
objects
    literal syntax, 32
optimal substructure, 153
optional chaining, 33
oscillator, 98
overlapping subproblems, 153

packed arrays, 28
partial application, 161
performance
    loops, 41
Picture-in-Picture, 100
pipe, 161
pop, 28
postorder, 146
preorder, 146
preventDefault, 60
private fields
    in operator, 118
Promise.all, 168
Promise.allSettled, 168
Promise.any, 168
Promise.race, 168
promises, 166
property descriptors, 35
prototype chain, 36, 121
prototype inheritance, 39
prototypes, 36
pure functions, 159
push, 28

quadratic complexity, 41
quick sort, 138
QuotaExceededError, 111

random numbers, 16
RangeError, 127
real-time validation, 86
recursion, 124
    nested data, 126
    vs iteration, 128
recursive step, 124
reduce, 53
    initial value, 53
references, 34
reflow, 59
regex
    anchors, 79

character classes, 79
flags, 79
lookahead, 81
lookbehind, 81
named groups, 80
performance, 81
quantifiers, 79
syntax, 78
regular expressions, 78
repaint, 59
replace, 80
return, 21
root node, 145

scope, 23
semantic HTML, 64
separate chaining, 134
sessionStorage, 109
Set, 105
difference, 106
intersection, 106
union, 106
setters, 35, 119
shift, 28
short-circuit evaluation, 14
side effects, 159
slice, 29
sliding window pattern, 140
some, 54
sort, 54
Source Maps, 75
space optimization, 155
splice, 29
spread operator, 30
stack overflow, 127
stack trace, 71
static
private fields, 119
stopPropagation, 60
storage
comparison, 109
storage event, 113
storage quota, 111
string, 5
String methods
regex, 80
string methods, 8
super, 119

tabindex, 66
tabulation, 154
tail call optimization, 127
TCO, 127
template literal, 7
textContent, 58
this keyword, 33
throttle, 60
timestamps, 90
timezone offset, 91
top-down DP, 154
toReversed, 29
toSorted, 29, 54
toSpliced, 29
tree traversal, 126, 146
TreeNode class, 146
trees, 145
truthy, 14
try-catch-finally, 72
two-pointer technique, 139
type conversion, 18

unhandled rejection, 73
unique values, 105
unshift, 28
UTC, 91

V8 engine, 28
ValidityState, 85
var, 4
variable, 3
vertex, 148
video, 96

WCAG, 64
contrast ratios, 68
WeakMap, 106
WeakSet, 106
Web Audio API, 98
with, 29
worst-case complexity, 137

XSS prevention, 88

yield, 41